JULIANA EYNG

PROGRAMAÇÃO PARALELA APLICADA AO MÉTODO *N-Scheme* PARA SOLUÇÃO DE PROBLEMAS COM O MÉTODO DE ELEMENTOS FINITOS

Florianópolis
2012

UNIVERSIDADE FEDERAL DE SANTA CATARINA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Juliana Eyng

PROGRAMAÇÃO PARALELA APLICADA AO MÉTODO *N-Scheme* PARA SOLUÇÃO DE PROBLEMAS COM O MÉTODO DE ELEMENTOS FINITOS

Tese submetida ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Doutora em Engenharia Elétrica.
Orientador: Prof. Dr. João Pedro Assumpção Bastos.
Coorientador: Prof. Dr. Marcos Fischborn.

Florianópolis, Dezembro de 2012.

Juliana Eyng

PROGRAMAÇÃO PARALELA APLICADA AO MÉTODO *N-Scheme* PARA SOLUÇÃO DE PROBLEMAS COM O MÉTODO DE ELEMENTOS FINITOS

Esta Tese foi julgada adequada para obtenção do Título de Doutora em Engenharia Elétrica, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.

Florianópolis, 04 de dezembro de 2012.

_____
Prof., Patrick Kuo-Peng, Dr.
Coordenador do Programa de Pós Graduação em Engenharia Elétrica

_____
Prof. João Pedro Assumpção Bastos, Dr.
Orientador

_____
Prof. Marcos Fischborn, Dr.
Coorientador - UTFPR

Banca Examinadora:

_____
Prof. Nelson Sadowski, Dr.
UFSC

_____
Prof. Luiz Lebensztajn, Dr.
USP/SP

_____
Prof. Mário A. R. Dantas, Dr.
UFSC

_____
Prof. Marcelo G. Vanti, Dr.
FURB

AGRADECIMENTOS

Agradeço a DEUS, por ter me dado à vida e a saúde.

Aos meus pais Adolfo e Maria, pela dedicação e carinho que sempre tiveram, pelo apoio e pelo incentivo que me deram em momentos importantes da vida e que posso resumir em uma única palavra: AMOR.

Ao meu orientador, Prof. João Pedro Assumpção Bastos, que não mediu esforços para a realização deste trabalho.

Ao coorientador, Prof. Marcos Fischborn pelo apoio que sempre deu.

Ao Prof. Mário Dantas, pelas orientações dadas em momentos importantes.

Aos meus colegas de estudo do GRUCAD, em especial a Juliana, Bruno, Diego, Juliano e Túlio.

Aos amigos do INEP Roberto e Roniere pela amizade e pelo apoio.

Aos demais Professores do GRUCAD: Prof. Nelson Sadowski, Prof. Jhoe Batistela, Prof. Walter Carpes Júnior, Prof. Maurício Valência, Prof. Jean Leite e em especial o Prof. Patrick Kuo-Peng.

Ao meu colega e aluno Maginot Júnior pelo apoio e dedicação que teve na fase final deste trabalho.

A *Sharcnet* por ter possibilitado a execução dos programas em sua rede de supercomputadores.

Aos demais membros da banca: Prof. Luiz Lebensztajn, relator do trabalho e ao Prof. Marcelo Vanti.

RESUMO

Nesta tese é proposta uma nova técnica para resolução de problemas estáticos com o método de elementos finitos denominada *N-Scheme*. O método resolve problemas de elementos finitos sem a montagem do sistema matricial $Ax=b$. A técnica calcula os potenciais nos nós incógnitos de uma maneira muito mais simples que a técnica convencional. A montagem e a solução do sistema matricial são consideradas em um único procedimento e as operações são similares ao método de Gauss-Seidel com sobre-relaxação (ou Successive Over Relaxation - SOR) que fornece boa convergência. Contudo, o tempo computacional do método *N-Scheme* é maior quando comparado com a implementação do método clássico de elementos finitos, como o ICCG (Incomplete Choleski Conjugate Gradient). Uma possível forma de melhorar o tempo computacional do método *N-Scheme* é aplicar técnicas de programação paralela. Estudos realizados recentemente mostraram que o método dos Gradientes Conjugados aplicado juntamente com o método *N-Scheme* reduz o tempo computacional significativamente. Assim, o trabalho de pesquisa da tese tem como objetivo principal mostrar que o novo método *N-Scheme* associado com as técnicas de programação paralela oferecem ainda melhores tempos computacionais na resolução de problemas em elementos finitos envolvendo malhas 3D.

Palavras-chave: Método de Elementos Finitos (MEF). Método Iterativo de Gauss-Seidel. Programação Paralela.

ABSTRACT

In this thesis a new technique for solving static problems with the finite element method is called *N-Scheme*. The method solves finite elements problems without assembling the matrix system $Ax=b$. It calculates the node potential unknowns in a much simpler way than the traditional technique of finite elements. The assembling and solution of the matrix system are considered in a single procedure and operations are similar to the Gauss-Seidel method with over-relaxation (or Successive Over Relaxation – SOR) providing good convergence. However, the computational time of *N-Scheme* method is larger when compared with the classical implementation of finite element method, such as ICCG (Incomplete Choleski Conjugate Gradient). One possible way to improve the computational time of the *N-Scheme* method is to apply parallel programming techniques. Recently, studies have shown that the Conjugate Gradient method applied in conjunction with the *N-Scheme* reduces the computational time significantly. Thus, this thesis aims to show that the new *N-Scheme* method associated with parallel programming techniques offers a still better computational time to solve problems involving finite element 3D meshes.

Keywords: Finite Element Method (FEM). Gauss-Seidel Iterative Method. Parallel Programming.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

1.1 MOTIVAÇÃO

Dado que muitos fenômenos físicos são representados por equações diferenciais, estudos são destinados à solução desses problemas de forma computacional, pois a solução dos mesmos nem sempre pode ser obtida de forma analítica. Assim, é preciso aplicar aproximações numéricas para obter um problema solúvel computacionalmente tal que o método numérico escolhido tenha boas propriedades de estabilidade, consistência e principalmente convergência. Para tanto, métodos numéricos são estudados e aplicados na solução desses problemas para obter-se precisão nos resultados, considerando também o custo computacional envolvido. Dentre aplicações que exigem métodos numéricos de elevada acurácia, estão aquelas modeladas pelas equações de Maxwell, que descrevem o comportamento de dispositivos eletromagnéticos, motivo de estudos deste trabalho [1].

Atualmente, dispõe-se de modelos físico-matemáticos e sistemas de cálculo em computadores bem estabelecidos com os quais é possível estudar, projetar e otimizar o comportamento desses dispositivos utilizando os recursos computacionais existentes. A necessidade de explorar esses recursos torna-se cada vez mais necessária, pois a evolução dos problemas de engenharia, motivada por fatores diversos tais como diminuição do consumo de energia, custos de produção, melhoria de materiais construtivos e competitividade entre fabricantes na disputa entre novos consumidores, vem acompanhada de uma necessidade crescente de refinamento dos modelos.

Neste contexto de solução de problemas eletromagnéticos, diversas classes de métodos têm ganhado destaque, citando-se o de elementos finitos, dos momentos, das diferenças finitas, entre outros. Especialmente neste trabalho de tese, tem interesse nos cálculos de problemas eletromagnéticos de baixa frequência, o método mais utilizado é o de elementos finitos.

Diversos modelos de dispositivos eletromagnéticos exigem o estudo e o cálculo criterioso de campos elétricos e/ou magnéticos realizados numericamente usando o método de elementos finitos (MEF), pois este possui vantagens computacionais em problemas envolvendo geometrias mais complexas [2]. O método consiste na discretização do domínio em um determinado número de regiões elementares, os elementos finitos, possuindo nós e arestas, onde a coleção de nós e

elementos são conhecidos como “malha de elementos finitos”. A partir das equações diferenciais que descrevem o comportamento de dispositivos eletromagnéticos, reformula-se o problema em um problema variacional em espaços de dimensão finita, onde representa-se a aproximação em termos de certas funções de base polinomiais (geralmente linear ou quadrática). Tradicionalmente, o procedimento numérico é estabelecido tanto através do “método variacional” como pelo “método de resíduos ponderados”, que, sob certas condições, é chamado “método de Galerkin” [3] [4] [5]. A solução do sistema pode ser obtida analisando-se os nós ou as arestas, dando origem a dois modos diferentes de obtenção dos resultados. Este problema variacional discreto conduz à necessidade de resolução de sistemas lineares esparsos cuja dimensão pode, em alguns casos, representar um impedimento na análise do problema.

A solução dos sistemas lineares gerados pelo método de elementos finitos pode ser realizada utilizando métodos diretos ou métodos iterativos. A escolha pelo método de resolução mais apropriado depende, principalmente, do número de incógnitas e da esparsidade da matriz do sistema linear a ser resolvido.

Assim, dependendo do modelo a ser estudado e das características do problema, as simplificações não serão permitidas e os recursos computacionais podem se mostrar extremamente limitados. Os limites podem ser físicos, como a quantidade de memória requerida e o tempo de cálculo, que depende do problema e do processo de resolução como um todo (método numérico utilizado, precisão, convergência, etc).

Tradicionalmente, utilizam-se os métodos diretos como a Eliminação de Gauss com pivotamento parcial ou a fatoração LU. Este método, na versão por banda, prevê os espaços de enchimento da matriz na hora do armazenamento. O pivotamento é adotado para minimizar problemas de arredondamento. Por serem métodos eficazes, fornecem solução de qualquer sistema, desde que ela exista.

Já os métodos iterativos estão relacionados a uma variedade de técnicas que usam aproximações sucessivas para obter a solução mais precisa, a cada passo, para os sistemas lineares e convergem para a solução sob certas condições. Os métodos iterativos podem ser mais rápidos que os métodos diretos e necessitam de menos memória do computador, mas quando aplicados na solução de problemas mais complexos podem consumir muito tempo de processamento na atualização de vetores, produtos matriz vetor e no cálculo de pré-condicionadores. Alguns destes processos de cálculo que demandam maior tempo de processamento podem ser paralelizados tanto em

arquiteturas com memória distribuída ou com memória compartilhada. Entretanto, a aplicação de programação paralela não implica em melhoria de desempenho.

Assim, com as dificuldades encontradas na resolução de sistemas lineares gerados pelo método de elementos finitos, constantes estudos são realizados para a obtenção de melhores resultados [6] [7] [8] [9] [10] [11]. Com isso, e tendo disponível as novas tecnologias e paradigmas de programação existentes, surge uma nova proposta na solução de problemas envolvendo sistemas lineares, denominado de *N-Scheme*. Trata-se de um método iterativo que propõe a solução de um sistema $Ax=b$ de maneira diferente da convencionalmente utilizada. Com esta técnica, a montagem e o armazenamento da matriz A não é mais necessária.

O texto que segue é motivo de estudo para este trabalho de tese que descreve o desenvolvimento desta nova técnica de resolução de sistemas lineares, *N-Scheme*, em conjunto com as técnicas de programação paralela em máquinas do tipo *cluster* e em máquinas multiprocessadas, aplicada ao método de elementos finitos cujas matrizes não necessitam de armazenamento.

1.2 POSICIONAMENTO DO PROBLEMA

O GRUCAD (Grupo de Concepção e Análise de Dispositivos Eletromagnéticos) desenvolveu ferramentas de cálculo eletromagnético bidimensional (EFCAD) e tridimensional (FEECAD) utilizando o método numérico de elementos finitos (EF). Esses são estruturados em três etapas: pré-processamento, processamento e pós-processamento. Cada etapa apresenta características e problemas particulares como, no processamento e na elaboração global de um sistema de cálculo. Por isso, o desenvolvimento de cada programa deve levar em consideração os seguintes aspectos: velocidade no processo de cálculo e a utilização da memória.

Na solução de problemas eletromagnéticos que apresentam uma geometria predominante em duas dimensões, uma análise bidimensional (2D) é suficiente e fornece resultados satisfatórios. Problemas eletromagnéticos em 2D são utilizados como testes na validação dos resultados, pois suas formulações são bem determinadas.

No entanto, aplicações do método em problemas eletromagnéticos tridimensionais (3D) é ainda uma área aberta a pesquisas. As dificuldades envolvendo o cálculo 3D são a motivação

para a contínua pesquisa na área, resultando em uma variedade de formulações que fornecem excelentes resultados em muitos problemas práticos [12]. Para auxiliar na resolução destes problemas, o laboratório adquiriu em 2007 o software I-DEAS (NX – IDEAS, comercializado pela empresa Siemens), que permite o desenho de dispositivos com geometrias mais complexas. Entretanto, esse software não foi concebido para realizar cálculos eletromagnéticos e é então, utilizado como pré e pós-processador. No presente trabalho, a ideia inicial era utilizar como dados de entrada, malhas de elementos finitos geradas pelo EFCAD, pelo FEECAD e pelo I-DEAS para diferentes geometrias.

Nas primeiras execuções do programa com a implementação em paralelo do método, utilizou-se as malhas geradas por estes programas. No entanto, as malhas precisam ter um tamanho considerável para justificar o uso da programação paralela. Assim, as maiores malhas geradas pelos mesmos, não eram suficientemente grandes para demonstrar o resultado desejado. Sendo o principal objetivo do trabalho mostrar que o método *N-Scheme* paralelizado oferece bons resultados em tempo de processamento comparado com o método em sequencial, optou-se por trabalhar com uma malha regular gerada no próprio programa, utilizando os mesmos princípios do malhador do FEECAD.

Por esses motivos e pela constante busca por novas soluções para os problemas, procura-se com este trabalho resolver problemas associados com a etapa de processamento. Ou seja, investiga-se uma solução alternativa para a resolução de sistemas lineares gerado pelo MEF, o método *N-Scheme*, uma solução diferente da convencionalmente utilizada. Com os resultados desse novo modelo associado às ferramentas de programação paralela, busca-se rápida convergência, consistência, estabilidade e baixo custo computacional na solução de sistemas lineares.

O eletromagnetismo pode ser dividido basicamente em domínios das baixas frequências (eletrotécnica) e domínios de altas frequências (ondas). O método será aplicado na resolução de problemas nos domínios das baixas frequências, que compreende a maior parte da faixa de operação de dispositivos eletromagnéticos, com ênfase em problemas eletrostáticos 3D.

Embora algumas formulações para resolução de problemas eletromagnéticos em 3D sejam muito utilizadas, novas aproximações numéricas e físicas ainda são motivos de estudo, como será apresentado nesse trabalho de tese.

1.3 OBJETIVOS

1.3.1 Objetivo Geral

O objetivo principal deste trabalho é acrescentar ao conjunto de softwares existentes no GRUCAD uma nova técnica de resolução de sistemas lineares denominada de *N-Scheme*, utilizando as vantagens de um paradigma de programação paralela na análise de dispositivos eletromagnéticos, por troca de mensagens e *multicore*. A ideia é trabalhar com um *cluster*, ou seja, um agregado de computadores que trabalham de maneira conjunta e/ou com máquinas multiprocessadas.

1.3.2 Objetivos Específicos

- Apresentar a nova técnica proposta (*N-Scheme*) na resolução de sistemas lineares $Ax=b$ advindos do método de elementos finitos utilizando os paradigmas de programação paralela por troca de mensagens e *multicore*;
- Implementá-la utilizando o método de programação paralela em uma linguagem de programação (FORTRAN 77 e C);
- Resolver grandes desafios computacionais reduzindo o tempo de execução em paralelo, comparado o tempo de execução em sequencial;
- Utilizar um agregado de computadores, os *clusters*, e máquinas *multicore* para execução dos programas;
- Realizar testes comparando as soluções numéricas e os desempenhos de tempo de execução entre os programas em sua forma sequencial e em paralelo;
- Disponibilizar o método paralelizado ao conjunto de softwares já existentes no GRUCAD;
- Publicar os resultados em conferências e periódicos da área.

Os estudos sobre os métodos computacionais paralelizados aplicados a elementos finitos na resolução de problemas eletromagnéticos podem ser justificados quando sua aplicação aumenta o desempenho no processamento (reduzindo significativamente o tempo) e apresenta-se a necessidade de resolver grandes desafios computacionais [13]. Mas alguns aspectos devem ser levados em consideração quando se pretende aplicar processamento paralelo na solução de problemas, como, por exemplo, o tempo de comunicação

dispendido entre os diversos processadores [14], assim como problemas com alocação de memória.

1.4 PROPOSTAS DA TESE E SUAS CONTRIBUIÇÕES

O início deste trabalho de tese teve como ponto de partida o estudo do funcionamento do método *N-Scheme*. Estes estudos estavam relacionados às características computacionais que o método apresenta quando comparado com o tradicionalmente utilizado na resolução de problemas com o MEF. Características como tempo de processamento, necessidade de armazenamento de vetores e matrizes são considerados aqui, com possíveis problemas com alocação de memória, número de iterações e convergência do método. Inicialmente foi estudada a aplicação do método em problemas 2D para diferentes geometrias com malhas geradas no EFCAD para analisar o comportamento do mesmo na solução dos problemas. Assim feito, passou-se para o estudo de problemas 3D, utilizando malhas geradas no FEECAD e no I-DEAS.

A introdução do método *N-Scheme* foi apresentada em [1] e pode-se observar que o método é eficiente e sua implementação é muito mais simples quando comparado ao método clássico de EF. Neste novo método, a montagem e a solução do sistema $Ax=b$ são considerados em um único procedimento, sendo desnecessária a alocação de memória para o sistema matricial. O procedimento resultante é iterativo e as operações realizadas são idênticas ao método de Gauss-Seidel. Técnicas de relaxação são aplicadas no método para acelerar a convergência, denominado método SOR (Successive Over Relaxation, ou seja, Gauss-Seidel com sobre-relaxação) [15].

Fazendo a comparação, o MEF clássico requer o cálculo das matrizes de contribuição dos elementos – matriz de rigidez (A) e vetor fonte (b). A construção de cada matriz é normalmente realizada elemento por elemento e a montagem do sistema matricial global denominado $Ax=b$. Condições de contorno são adicionadas e o sistema é resolvido por um método direto, como Eliminação de Gauss, ou por um método iterativo, dos Gradientes Conjugados (GC) com Decomposição Incompleta de Cholesky (IC), o ICCG. Para a solução, são necessários os passos de montagem, armazenamento e resolução do sistema, o que para o *N-Scheme* não é mais necessário.

Entretanto, o método *N-Scheme* apresenta um tempo computacional na resolução dos problemas que é maior quando comparado com o MEF clássico como o ICCG. Assim, para melhorar a

velocidade de processamento da nova técnica, uma possível solução é a aplicação de programação paralela [16].

Outra solução também estudada e apresentada em [17], aplica o método GC associado com o pré-condicionador de Jacobi (GC-J) mantendo os princípios do método *N-Scheme*. Como resultado, temos uma redução significativa no tempo de processamento no algoritmo em sua forma sequencial [18] [19] [20]. Aqui também foram realizados estudos para melhorar ainda mais este tempo utilizando o processamento paralelo.

Antes de explorar o processamento paralelo do método, pesquisou-se sobre métodos iterativos estacionários como Gauss-Seidel sob a abordagem de programação paralela. Esse método apresenta nível de granulação muito fino. Granulação refere-se à forma de como uma tarefa é paralelizada, indica muitas tarefas pequenas com muita comunicação [13] [14], que será abordado com mais detalhes no capítulo 3.

A utilização do processamento paralelo implica a utilização de máquinas com arquitetura afim. Este trabalho utilizou *clusters* de computadores, uma arquitetura de memória distribuída, cujos nós são conectados por uma rede local de alta velocidade, onde a comunicação entre os nós acontece através da passagem de mensagens. As execuções dos programas foram realizadas no *cluster* de computadores da *Sharcnet*[1] (Shared Hierarchical Academic Research Computing Network) localizado no Canadá. Através do Prof. Dr. Mário Antônio Ribeiro Dantas, do curso de Ciências da Computação da UFSC, foi possível esse intercâmbio para criação de uma conta particular e utilização do sistema de alto desempenho da *Sharcnet* de forma remota. A *Sharcnet* possui sistemas de *clusters* com diferentes tipos de redes de comunicação, podendo-se assim realizar experimentos para diferentes conjuntos de *clusters*. Um dos motivos que nos levou a utilizar a rede de computadores da *Sharcnet* foi evitar possíveis dificuldades quando da instalação de um *cluster* próprio.

Com a modificação do método *N-Scheme* com GC que o tornou mais rápido em sua forma sequencial, trabalhou-se também com a ideia de programação paralela em ambientes multiprocessados.

A evolução do trabalho resultou na necessidade da criação de malhas de EF com dimensões maiores das que foram conseguidas pelos pacotes de softwares existentes no GRUCAD. Para problemas de pequena dimensão, a utilização de métodos iterativos não se justifica,

[1] SHARCNET homepage, http:/www.SHARCNet.ca

uma vez que os métodos diretos são comprovadamente mais rápidos e eficientes nestas condições. Assim, com base na técnica utilizada na geração de malhas do FEECAD, criou-se um procedimento semelhante no próprio algoritmo desenvolvido neste trabalho, para criação das malhas e obtenção das informações necessárias para implementação e para a realização dos experimentos.

Como o método proposto não necessita de armazenamento de matrizes, os esforços se concentraram em como o problema seria dividido para a aplicação da programação paralela de forma eficiente. Uma das estratégias de programação paralela aplicada em conjunto com o método *N-Scheme* utiliza comunicação por troca de mensagens em *clusters* de computadores e outra utiliza estratégias de programação paralela em ambientes multiprocessados.

Inicialmente, considerou-se que as principais ideias eram originais desde o seu desenvolvimento e implementação do método *N-Scheme*. A proposta foi colocada pelo grupo de pesquisa do GRUCAD, sem qualquer apoio de trabalhos anteriores. Prosseguiu-se com uma pesquisa bibliográfica e não se encontrou qualquer trabalho semelhante. O assunto também foi discutido com colegas que têm uma longa experiência com técnicas de EF e chegou-se a uma conclusão idêntica. E, portanto, o grupo publicou três trabalhos sobre o assunto [1] [15] [17]. No entanto, mais tarde, foi revelado que alguns artigos já haviam sido publicadas sobre o tema [21] [22]. Menciona-se este fato por causa do respeito científico para trabalhos anteriores e seus autores. A importância em apresentar esta técnica está nas poucas bibliografias econtradas sobre o assunto e, além disso, há um interesse acadêmico e prático.

Embora exista uma grande bibliografia incluindo livros e artigos sobre o estudo do processamento paralelo aplicado à resolução de sistemas de equações, o assunto ainda necessita de mais publicações. Portanto, neste texto de tese, a maior contribuição é à aplicação de processamento paralelo no novo método de resolução de sistemas de equações em sistemas de elementos finitos, denominado *N-Scheme*, aplicado a problemas em 3D.

Assim, procurou-se estabelecer algumas diretrizes para este trabalho, tentando responder e justificar a utilização de plataformas de alto desempenho e o processamento paralelo na resolução de sistemas de equações gerado pelo MEF. Além disso, pretende-se deixar contribuições que possam auxiliar trabalhos futuros nesta área.

Algumas diretrizes estabelecidas no desenvolvimento deste trabalho:

- Existem diferenças de desempenho do algoritmo executado na forma sequencial e na forma paralela? A implementação do método *N-Scheme* utilizando processamento paralelo é adequado? Existe uma forma mais eficiente?
- Qual deve ser a dimensão do problema para que se tenha um ganho de tempo na execução do algoritmo em paralelo?
- Este trabalho propõe uma nova maneira de resolver um sistema de equações lineares que é claramente mais simples que a tradicionalmente utilizada. Em função do ganho de desempenho encontrado com a paralelização do método, qual a perspectiva de utilização de *clusters* de computadores? E a utilização de máquinas multiprocessadas? Qual seria o próximo passo na pesquisa envolvendo o processamento paralelo?
- A técnica de paralelização utilizada para o método é válida? É a única forma de paralelizar? Que outras abordagens poderiam ser estudadas pelo grupo de pesquisa no campo de processamento paralelo?

Os comentários sobre as diretrizes estabelecidas nesta seção serão discutidas com mais detalhes no capítulo 6.

1.5 ESTRUTURA DO TEXTO

Considerando-se os objetivos deste trabalho, o texto foi dividido em seis capítulos.

No primeiro capítulo, procurou-se apresentar uma descrição geral sobre o assunto, incluindo a motivação, o posicionamento do problema junto ao grupo de pesquisa GRUCAD, os objetivos e sua principal contribuição.

No segundo capítulo são apresentados o método de elementos finitos (MEF) e o novo método proposto (*N-Scheme*) para resolução do sistema matricial $Ax=b$, no qual o presente trabalho tem especial interesse.

O terceiro capítulo traz conceitos de computação paralela e da arquitetura de computadores existentes e voltados para *clusters*. Este capítulo ainda inclui uma descrição do paradigma de programação paralela por troca de mensagens, apresentando conceitos e rotinas de comunicação *Message Passing* e estratégias de paralelização em ambientes multiprocessados. Apresenta-se também, conceitos e

características da programação paralela em memória distribuída e compartilhada,

No quarto capítulo são apresentados os procedimentos adotados para paralelizar o método *N-Scheme-SOR* e o *N-Scheme-GC*.

O capítulo cinco mostra os ambientes utilizados nas execuções dos programas e os resultados alcançados nos casos estudados. São apresentados gráficos e tabelas comparativas mostrando o tempo de execução do método *N-Scheme-SOR* e *N-Scheme-GC* em sequencial e em paralelo.

No sexto capítulo faz-se uma conclusão geral sobre a aplicação do processamento paralelo no método *N-Scheme*, juntamente com as resposta para as diretrizes estabelecidas do início do trabalho em função dos resultados encontrados.

2 O MÉDOTO DE ELEMENTOS FINITOS E A NOVA TÉCNICA *N-Scheme*

Neste capítulo será apresentado brevemente o método clássico de elementos finitos, consagrado há várias décadas. Será abordado também, o novo método *N-Scheme* para resolver o sistema matricial $Ax=b$, tema fundamental deste trabalho de tese. A técnica *N-Scheme* se utiliza de um processo de resolução mais simples e não se tem mais interesse na forma das matrizes geradas pela discretização do MEF. Conceitos de eletromagnetismo e os métodos numéricos utilizados na resolução de problemas eletromagnéticos serão brevemente abordados.

2.1 EQUACIONAMENTO BÁSICO DE ELETROMAGNETISMO

As Equações de Maxwell são um grupo de quatro equações, assim denominadas em decorrência dos trabalhos de James Clerk Maxwell e que descrevem o comportamento dos campos elétricos e magnéticos. Estas leis, em adição com as equações constitutivas (iterações com a matéria) regem os mais diversos fenômenos do eletromagnetismo [23]. Para um caso geral, as quatro leis de Maxwell são assim apresentadas:

Equações de Maxwell:

$$rot\ \vec{H} = \vec{J} + \frac{\partial \vec{D}}{\partial t} \tag{2.1}$$

$$div\ \vec{B} = 0 \tag{2.2}$$

$$rot\ \vec{E} = -\frac{\partial \vec{B}}{\partial t} \tag{2.3}$$

$$div\ \vec{D} = \rho \tag{2.4}$$

Leis Constitutivas:

$$\vec{B} = ||\ \mu\ ||\ \vec{B} \tag{2.5}$$

$$\vec{D} = ||\ \varepsilon\ ||\ \vec{E} \tag{2.6}$$

$$\vec{J} = ||\ \sigma\ ||\ \vec{E} \tag{2.7}$$

As grandezas físicas que aparecem nas equações de Maxwell são: o campo magnético $\vec{H}$ [A/m], a indução magnética ou densidade de fluxo magnético $\vec{B}$ [T], o campo elétrico $\vec{E}$ [V/m], a indução elétrica ou densidade de fluxo elétrico $\vec{D}$ [C/m^2], a densidade superficial de corrente $\vec{J}$ [A/m^2] e a densidade volumétrica de carga ρ [C/m^3]. Essas grandezas são, exceto ρ, campos vetoriais.

As equações de Maxwell com as relações constitutivas (2.5) à (2.7), formam um sistema de equações que deve ser resolvido de forma simultânea. As características físicas dos materiais são representadas pelos seguintes tensores:

$$|| \mu || = \begin{bmatrix} \mu_{x,x} & \mu_{x,y} & \mu_{x,z} \\ \mu_{y,x} & \mu_{y,y} & \mu_{y,z} \\ \mu_{z,x} & \mu_{z,y} & \mu_{z,z} \end{bmatrix}, \; || \varepsilon || = \begin{bmatrix} \varepsilon_{x,x} & \varepsilon_{x,y} & \varepsilon_{x,z} \\ \varepsilon_{y,x} & \varepsilon_{y,y} & \varepsilon_{y,z} \\ \varepsilon_{z,x} & \varepsilon_{z,y} & \varepsilon_{z,z} \end{bmatrix}, \; e \; || \sigma || = \begin{bmatrix} \sigma_{x,x} & \sigma_{x,y} & \sigma_{x,z} \\ \sigma_{y,x} & \sigma_{y,y} & \sigma_{y,z} \\ \sigma_{z,x} & \sigma_{z,y} & \sigma_{z,z} \end{bmatrix} \quad (2.8)$$

O conjunto das equações apresentadas em (2.1) à (2.7), permitem o equacionamento e a resolução da maior parte dos problemas envolvendo grandezas eletromagnéticas.

As equações de Maxwell podem ser aplicadas a diversas situações particulares e para cada uma delas as características do problema devem ser adotadas de tal forma que o equacionamento resultante seja coerente com as condições do caso em questão [23].

2.1.1 O potencial escalar elétrico

O uso de potenciais é muito frequente e conveniente na resolução de problemas de campos. O potencial escalar elétrico possibilita a determinação de campos elétricos em problemas estáticos. Neste trabalho, optou-se por estudar os campos elétricos criados por cargas ou variações de potenciais elétricos, o que, agregado de alguns conceitos complementares constitui a Eletrostática [23]. Suas equações básicas são (2.4) e (2.6):

$div \, \vec{D} = \rho$, oriunda do grupo das quatro equações de Maxwell, Equação (2.4) e $\vec{D} = || \varepsilon || \, \vec{E}$, relação constitutiva onde a permissividade do material é presente, Equação (2.6).

Supõe-se que no domínio de estudo não há variação temporal de grandezas. Neste caso, podemos definir um potencial escalar *V* do qual deriva o campo eletrostático conservativo [23]:

$$\vec{E} = -grad\ V \tag{2.9}$$

Assumindo ρ=0, no caso aonde não há cargas estáticas e que só exista um meio dielétrico no domínio de estudo, ε constante, (isotrópico) tem-se:

$$\vec{D} = \varepsilon \vec{E} \tag{2.10}$$

$$div\ \varepsilon \vec{E} = 0 \tag{2.11}$$

$$div\ \varepsilon(-grad\ V) = 0 \tag{2.12}$$

que, de forma explícita, é:

$$\frac{\partial^2 V}{\partial x^2} + \frac{\partial^2 V}{\partial y^2} + \frac{\partial^2 V}{\partial z^2} = 0 \tag{2.13}$$

A Equação (2.13) é uma equação de Laplace definindo a distribuição de *V* em meios homogêneos. Neste caso, a fonte geradora de campo elétrico advém obrigatoriamente das condições de contorno pela imposição de diferenças de potencial.

2.2 O MÉTODO DE ELEMENTOS FINITOS

O MEF é um método numérico que consiste na discretização de um problema físico em um número finito de pequenas regiões denominadas de “elementos finitos”. A essa divisão de domínio se dá o nome da malha que mantém as mesmas propriedades físicas do meio original. O MEF é considerado um método matemático para resolução de equações diferenciais e sua modelagem é normalmente feita por equações diferenciais ou integrais com suas respectivas condições de contorno [3] [4] [5]. Devido às suas características de flexibilidade e estabilidade numérica, o MEF pode ser implementado na forma de um

sistema computacional de maneira consistente e sistemática, o que explica sua grande popularidade.

Em muitos casos práticos, o MEF é a única ferramenta capaz de fornecer uma solução aceitável, ainda que, sob o ponto de vista matemático, a solução seja considerada uma aproximação numérica.

O MEF é amplamente utilizando para resolver problemas eletromagnéticos como transformadores, motores, contatores, etc., permitindo o cálculo de potenciais e campos elétricos e magnéticos do problema [2] [12].

Tradicionalmente, o procedimento numérico é estabelecido tanto através do "método variacional" como pelo "método dos resíduos ponderados", que sob certas condições é chamado "método de Galerkin". Neste trabalho, tem-se interesse particular neste último que é muito utilizado em Eletromagnetismo.

Formulações 2D para o MEF são utilizadas para validação de resultados por ser didaticamente mais simples de compreender. Mas, no âmbito deste trabalho, o interesse maior está nos problemas 3D que costumam gerar sistemas matriciais de ordem elevada. A resolução do sistema matricial $Ax=b$ é a etapa que leva mais tempo para ser executada quando implementada computacionalmente.

Para solucionar o problema da resolução de sistemas lineares para grandes malhas de EF, surge a necessidade de encontrar uma forma de torná-lo propício à resolução, pesquisando por novas técnicas de programação, como a aplicação de programação paralela e/ou por uma nova técnica de solução do mesmo. É neste aspecto que se situa a proposta do trabalho desta tese.

2.3 O MÉTODO *N-Scheme* NA FORMA SEQUENCIAL

A solução de problemas eletromagnéticos utilizando o MEF resulta na solução de um sistema matricial $Ax=b$. O método clássico utilizado na resolução desse sistema necessita da montagem e armazenamento do sistema matricial $Ax=b$. O novo método *N-Scheme* de solução de $Ax=b$ [1], é considerado muito mais simples que o tradicional e prova que a montagem da matriz A não é mais necessária. A diferença entre o método proposto em [1] e o convencional está no consumo de tempo de processamento. A nova técnica requer mais iterações para convergir, embora os resultados sejam bastante confiáveis.

A técnica proposta trabalha com os nós e as células de elementos que rodeiam os nós, ou seja, elementos que tenham como vértice o nó *n*. O método é resolvido basicamente da forma como segue:

- Para cada nó incógnita da malha:
  - Calcula-se a contribuição de cada elemento da célula que rodeia esse nó;
  - Tem-se a primeira aproximação do potencial para esse nó incógnita;
- Repete-se o processo com os novos valores de potenciais calculados dos nós até que a convergência seja atingida.

Pode-se notar que neste método, os potencias são calculados sem a montagem, armazenamento e resolução do sistema $Ax=b$. A solução é implícita no próprio esquema. Como é um método iterativo, uma questão importante é a convergência. Para que o método alcance a convergência mais rapidamente, é adicionado um fator de relaxação na solução.

Considere a malha 2D da figura abaixo:

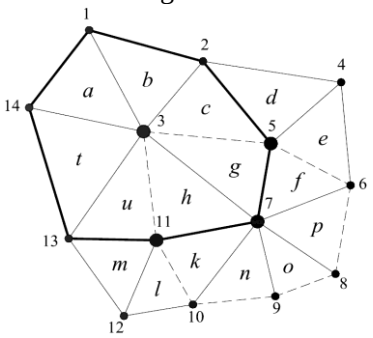

Figura 1– Malha de EF 2D de elementos triangulares [1].

Na Figura 1, os nós são indicados por números e enquanto os elementos por letras. Suponha que os nós 3, 5, 7 e 11 são incógnitas e que os outros (1, 2, 4, 6, 8, 9, 10, 12, 13 e 14) possuem valores de condições de contorno impostos. Suponha que o potencial incógnita é denominado *v* e que o elemento *a* é o único a atuar no nó 3. Se a numeração dos nós criada pela malha possui a sequencia 14, 3 e 1, a matriz elementar dada por esse elemento é:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,1} & a_{3,2} & a_{3,3} \end{bmatrix} \begin{bmatrix} v_{14} \\ v_3 \\ v_1 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \end{bmatrix} \tag{2.14}$$

Supondo que os valores de v_1 e v_{14} são conhecidos (como condições de contorno de Dirichlet), pode-se escrever, para o potencial v_3:

$$a_{2,2} v_3 = s_2 - a_{2,1} v_{14} - a_{2,3} v_1 \tag{2.15}$$

Se a sequência é 3, 14 e 1, tem-se a expressão equivalente:

$$a_{1,1} v_3 = s_1 - a_{1,2} v_{14} - a_{1,3} v_1 \tag{2.16}$$

E para a sequência 1, 14 e 3, tem-se:

$$a_{3,3} v_3 = s_3 - a_{3,1} v_1 - a_{3,2} v_{14} \tag{2.17}$$

Um dos sistemas acima (Equações (2.15), (2.16) ou (2.17)) deve interagir com outros elementos que rodeiam o nó 3 e cada equação é calculada para todos os elementos dessa célula. Em outras palavras, o potencial calculado no nó 3 é obtido através da contribuição de cada elemento que rodeia esse nó que gera um sistema como mostrado na Equação (2.14). Para generalizar, *n* é o nó incógnita e *j* e *l* os outros 2 nós do elemento (como o nós 1 e 14 para o elemento *a*). Assim, faz-se a soma de todos os termos da diagonal desses sistemas relacionadas ao nó 3.

A soma do lado direito é definida por:

$$r_{\mathrm{j,l}}^{elem} = s_{\mathrm{n}} - a_{\mathrm{n,j}} v_{\mathrm{j}} - a_{\mathrm{n,l}} v_{\mathrm{l}} \tag{2.18}$$

A soma dos termos da diagonal:

$$diag\_sum = a_{\mathrm{n,n}}^{a} + a_{\mathrm{n,n}}^{b} + a_{\mathrm{n,n}}^{c} + a_{\mathrm{n,n}}^{g} + a_{\mathrm{n,n}}^{h} + a_{\mathrm{n,n}}^{u} + a_{\mathrm{n,n}}^{t} \tag{2.19}$$

Para o lado direito:

$$right\_sum = r_{j,l}^{a} + r_{j,l}^{b} + r_{j,l}^{c} + r_{j,l}^{g} + r_{j,l}^{h} + r_{j,l}^{u} + r_{j,l}^{t} \qquad (2.20)$$

Obtendo o valor de v_3 por:

$$v_3 = \frac{right\_sum}{diag\_sum} \qquad (2.21)$$

Operações similares são feitas para os nós incógnitas v_5, v_7 e v_{11}. No início do processo iterativo, os potenciais nos nós incógnitas são impostos. Assim feito, tem-se uma primeira aproximação para os nós incógnitas da malha utilizada como exemplo (Figura 1).

A próxima iteração inicia com os valores dos nós conhecidos e calculados na iteração anterior. Assim o processo é realizado até que a convergência seja alcançada como mostra o algoritmo da Figura 2.

$v = 0$ *incógnitas*
$v = V_{imposto}$ *condições de contorno*
for i = 1, 2, ... *loop das iterações*
  for n=1, nº nós
    diag_sum = 0
    right_sum = 0
    for j=1, nº elem
      iel = elemento(n, j)
      calcula a(3, 3) e s(3) para iel
      diag_sum = diag_sum + $a_{n,n}^{iel}$
      right_sum = right_sum + $r_{j,l}^{iel}$
    end for
    $v(n) = \frac{right\_sum}{diag\_sum}$
  end for
  verifica convergência
end for

Figura 2 – Algoritmo do método *N-Scheme* [1].

De acordo com o algoritmo da Figura 2, pode-se observar a implementação do método em sua forma sequencial. As operações algébricas realizadas pelo método são idênticas ao método de Gauss-Seidel [24] [25] [26]. A convergência com este método é lenta, sendo necessária a aplicação de um fator de sobre-relaxação para conseguir a solução mais rapidamente. Outro fator importante para que o método tenha convergência garantida, é que a matriz *A* deve ser diagonal dominante, o que significa que para cada linha o termo da diagonal principal seja, em módulo, maior ou igual à soma dos módulos dos demais termos dessa linha, garantido que, em pelo menos numa linha, o módulo seja maior. Essa condição é suficiente mas não necessária, podendo ocorrer convergência sem que a matriz seja diagonalmente dominante.

Para cada nó *n* é calculada a diferença entre o potencial v_n^{it}, da iteração atual (*it*) e o potencial v_n^{it-1}, da iteração anterior (*it-1*).

$$dif\ v_n = v_n^{it} - v_n^{it-1} \tag{2.22}$$

A aplicação do fator de relaxação é definida por:

$$v_n = v_n + (w-1)\ dif\ v_n \tag{2.23}$$

onde *w* é o fator de relaxação. Aqui, faz-se $R = w-1$, logo a expressão acima torna-se:

$$v_n = v_n + R\ dif\ v_n \tag{2.24}$$

se $R = 0$, não há relaxação;
para $0 < R < 1$, tem-se sobre-relaxação;
para $-1 < R < 0$, temos sub-relaxação.

Para os exemplos utilizados nos experimentos, é necessário utilizar um fator de relaxação que esteja entre $0{,}8 < R < 0{,}99$. Entretanto, o coeficiente de relaxação no EF não foi facilmente determinado, mas a expressão abaixo, obtida por [1], é eficiente:

$$R = R_{final}\left(1 - e^{-it/T}\right) \tag{2.25}$$

onde *it* é o número de iterações e um bom valor para $R_{final} = 0,96$ e para $T = 14.11$.

O fator de relaxação *R* é calculado de acordo com a iteração e tende à R_{final} à medida que o número de iterações aumenta. O fator *R* é aplicado no algoritmo após calcular os potencias nos nós *v(n)* como mostra a Figura 2. Como critério de parada para as iterações, analisa-se a convergência do método através do erro em cada nó.

Esta técnica pode ser utilizada em muitos casos, mas em particular, nos casos onde o sistema linear a ser resolvido é grande, como para malhas de EF 3D. Com as dificuldades encontradas na geração de malhas suficientemente grandes para aplicação do processo, optou-se por uma malha regular e uma geometria simples. Assim, o mesmo esquema de resolução adotado para malhas com elementos tetraédricos, foi adotado para uma malha com elementos hexaedros regulares.

2.3.1 O Método *N-Scheme* para uma Malha Regular

Considere as malhas das figuras abaixo:

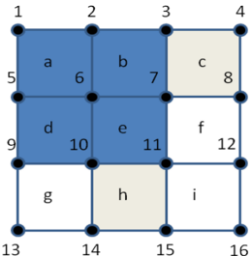

Figura 3 – Malha 2D de elementos retangulares.

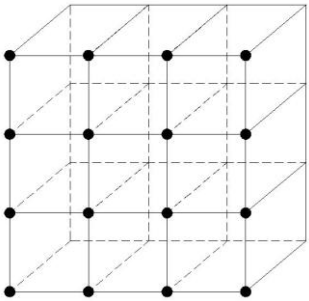

Figura 4 – Malha 3D de elementos hexaedros.

Como para os casos de malhas com elementos tetraédricos, utiliza-se aqui o caso da Figura 3, uma malha 2D, para explicar como ficam as equações do método *N-Scheme*. Os nós são indicados por números enquanto os elementos por letras. Suponha que os nós 6, 7, 10 e 11 são incógnitas e os outros nós possuem condições de contorno impostas.

Toma-se o nó 6, os elementos correspondentes são: *a*, *b*, *d* e *e*, formando sua célula. Calculando a matriz elementar do elemento *a*, obtêm-se uma matriz *a(4,4)*; para a fonte um vetor *s(4)*. Suponha que o potencial incógnita é chamado de *v* e que o elemento *a* é o único atuando sobre o nó 6. Se a numeração dos nós gerada pelo malhador tem a sequência 1, 5, 6 e 2, a matriz elementar deste elemento é dada por:

$$\begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} \end{bmatrix} \begin{bmatrix} v_1 \\ v_5 \\ v_6 \\ v_2 \end{bmatrix} = \begin{bmatrix} s_1 \\ s_2 \\ s_3 \\ s_4 \end{bmatrix} \tag{2.26}$$

Suponha que v_1, v_5 e v_2 são conhecidos (com condições de contorno de Dirichlet) pode-se escrever o potencial de v_6 como:

$$a_{3,3}v_6 = s_3 - \left(a_{3,1}v_1 + a_{3,2}v_5 + a_{3,4}v_2\right) \tag{2.27}$$

Neste caso o nó 6 é o terceiro na numeração do malhador, mas ele pode também ser o primeiro, o segundo ou o quarto e a expressão acima é adaptada. O sistema da Equação (2.26) vai interagir com os outros elementos que rodeiam o nó 6. Assim para cada elemento que contribui no cálculo do potencial do nó 6, formando a célula, a equação é calculada.

A soma de todos os termos da diagonal relacionadas ao nó 6:

$$diag\_sum = a^a_{n,n} + a^b_{n,n} + a^d_{n,n} + a^e_{n,n} \tag{2.28}$$

A soma do lado direto do sistema:

$$right\_sum = r^a_{j,l,k} + r^b_{j,l,k} + r^d_{j,l,k} + r^e_{j,l,k} \tag{2.29}$$

sendo *n* o nó incógnita, *j*, *l* e *k* os outros nós do elemento; para um elemento qualquer *elem*:

$$r_{j,l,k}^{elem} = s_n - a_{n,j}v_j - a_{n,l}v_l - a_{n,k}v_k \tag{2.30}$$

O nó 6 também interage com os nós 7, 10 e 11 através dos elementos comuns *b*, *d*, e *e*. Com os valores calculados nas Equações (2.28) e (2.29) pode-se obter o valor de v_6 por:

$$v_6 = \frac{right\_sum}{diag\_sum} \tag{2.31}$$

Operações similares são realizadas nos nós incógnitas (v_7, v_{10} e v_{11}). Como o método é iterativo, os potenciais são inicialmente impostos como zero. A próxima iteração inicia com os valores dos potenciais calculados na iteração anterior. O processo é executado até que a convergência seja alcançada.

Para os casos de malhas 3D, Figura 4, o procedimento é semelhante. A diferença é que se tem uma matriz de contribuição *a(8,8)* e o vetor fonte *s(8)* para cada elemento.

Como se pode observar, o algoritmo do método *N-Scheme* é muito mais simples que o método clássico de EF e sua principal contribuição é que os potenciais são calculados sem a montagem e o armazenamento de matrizes e vetores.

Como dito, as execuções utilizando o método em sua forma sequencial, mostram que o tempo computacional é maior quando comparado com a implementação clássica do MEF usando, por exemplo, o ICCG. O tempo computacional é o ponto do método *N-Scheme* que merece atenção. Uma possível forma de melhorar o desempenho do método é aplicar técnicas de paralelização.

Além disso, em [15] propõe-se algumas modificações no método *N-Scheme* em sua forma sequencial com o intuito de melhor o tempo de processamento. Depois de alguns esforços, notou-se que a aplicação do método de Gradiente Conjugado associado ao pré condicionador de Jacobi oferece ganhos significativos no tempo de processamento. A utilização desta técnica representa parte das contribuições inovadoras desta tese e o método *N-Scheme-GC* será apresentado com mais detalhes na próxima seção.

2.3.2 Adaptação do Método *N-Scheme* com Gradientes Conjugados

A diferença do método adaptado com o anteriormente descrito está no número de estruturas de repetição executadas (*loops*). No primeiro, são necessários três *loops*: o mais externo para as iterações, o segundo é para os nós incógnitas e o *loop* mais interno que está relacionado com a "célula" de elementos do nó incógnita que pode ser observado no algoritmo da Figura 2.

Na adaptação do método *N-Scheme-GC*, tem-se apenas dois *loops*: o mais externo para as iterações e o interno para os elementos. A Figura 5 mostra o esquema da adaptação, onde o método *N-Scheme* aparece em dois momentos no algoritmo, denominado *N-Scheme (1)* e *N-Scheme (2)*:

Calcular $r^{(0)} = b - Ax^{(0)}$ ⟶ *N-Scheme (1)*
for $i = 1, 2, ...$ ⟶ *Loop das iterações*
resolve $M\, z^{(i-1)} = r^{(i-1)}$
$\rho_{i-1} = r^{(i-1)^T} z^{(i-1)}$
if $i = 1$
$p^{(1)} = z^{(0)}$
endif
if $i > 1$
$\beta_{i-1} = \rho_{i-1} / \rho_{i-2}$
$p^{(i)} = z^{(i-1)} + \beta_{i-1}\, p^{(i-1)}$
endif
$q^{(i)} = A\, p^{(i)}$ ⟶ *N-Scheme (2)*
$\alpha_i = \rho_{i-1} / p^{(i)^T} q^{(i)}$
$x^{(i)} = x^{(i-1)} + \alpha_i p^{(i)}$
$r^{(i)} = r^{(i-1)} - \alpha_i q^{(i)}$
Verifica convergência
Continua de necessário
end

Figura 5 – Algoritmo do método *N-Scheme-GC*.

Uma breve explicação das variáveis e operações do algoritmo é necessária para um melhor entendimento do mesmo.

- *A* é a matriz de coeficientes;
- *b* é o vetor fonte (lado direito);
- *M* pode ser uma matriz pré-condicionadora;
- *z*, *r*, *p* são vetores que têm dimensões do tamanho do número de nós incógnitas;
- α e β são números escalares;
- $x^{(0)}$ é um conjunto de valores iniciais para a incógnita *x*.

No algoritmo apresentado na Figura 5 pode-se observar que o método *N-Scheme* é aplicado em duas operações. A primeira é em $r^{(0)} = b - Ax^{(0)}$ realizada uma única vez, não sendo necessária a aplicação de paralelização. Observando o lado direito da equação $b - Ax^{(0)}$, nota-se que ele pode ser divido em duas partes. Suponha que para a incógnita *n*, tem-se:

$$b_n - \sum_{j=1,N}^{N} a_{n,j} x_j \tag{2.32}$$

onde $n \neq j$ e, *n* é o número de incógnitas e $a_{n,j}$ é um termos genérico de *A*. E

$$a_{n,n} x_n \text{ , para os termos da diagonal} \tag{2.33}$$

A soma das Equações (2.32) e (2.33) produz o resíduo *r*. O resultado da Equação (2.32) é chamado *right_sum*. $a_{n,n}$ é calculado com o método *N-Scheme* e é chamado *diag_sum*. Para calcular o resíduo no início do processo iterativo, *diag_sum* é multiplicada por $x^{(0)}$ e subtraida de *right_sum*.

$$r^{(0)} = right\_sum - diag\_sum\, x^{(0)} \tag{2.34}$$

Isto corresponde a $r^{(0)} = b - Ax^{(0)}$.

A segunda operação onde aplicamos o método *N-Scheme* no algoritmo é em $q^{(i)} = Ap^{(i)}$, que é calculado de forma similar. Esta

operação é realizada dentro do *loop* das iterações, sendo aqui, aplicados os procedimentos da paralelização.

Para a matriz *M*, pré-condicionadora, utilizou-se o pré-condicionador de Jacobi, que é um vetor e que contém os elementos da diagonal principal da matriz *A*. Esta diagonal é calculada como *diag_sum* e a operação $M\, z^{(i-1)} = r^{(i-1)}$ pode ser facilmente realizada por:

$$z_n^{(i-1)} = r_n^{(i-1)} / diag\_sum_n \tag{2.35}$$

Ao realizar os *loops* sobre os elementos, estas matrizes elementares são avaliadas e usadas para calcular $q^{(i)} = A\, p^{(i)}$.

No algoritmo do método *N-Scheme* anteriormente descrito (*N-Scheme-SOR*), trabalha-se com os nós da malha de EF, exigindo assim um *loop* para os nós e outro para os elementos. Neste novo modelo, o método foi adaptado para uso com o método dos gradientes conjugados onde se trabalha com os elementos, ou seja, reduz-se a um *loop* mais interno apenas para os elementos.

A matriz *A* é referenciada apenas na forma de um produto matriz-vetor, ou a sua variante de transposição, em que *p* representa um vetor conhecido, cujo produto com *A* na iteração *it* é atribuído a um vetor *q*. Estes vetores são de tamanho *n* e são utilizados como pesquisa e atualização de vetores, em que *n* indica o número de graus de liberdade do sistema. A avaliação da Equação (2.30) pode ser realizado sem o cálculo explícito de *A*, realizando o produto numa base de elemento a elemento.

Esta abordagem exige, essencialmente, mais tempo de processamento em comparação com a estratégia convencional de montagem, mas que evita a formação explícita da matriz *A* e, como resultado disso, não requer mínima memória, pois as avaliações das matrizes elementares são realizadas durante as iterações.

O uso de pré-condicionadores é fundamental nos métodos iterativos não estacionários. Através do seu emprego se consegue uma convergência mais rápida. O pré-condicionamento não é aplicado diretamente sobre a matriz de coeficientes, mas aplicado através de uma relação envolvendo o resíduo da iteração. Existem vários tipos de pré-condicionadores, aqui, descreve-se a forma como os pré-condicionadores são inseridos nos algoritmos dos subespaços de Krylov.

A convergência pode ser melhorada empregando um precondicionador adequado, mas a aplicação de um pré-condicionador sem violar o princípio básico de um cálculo matricial-livre, isto é, sem explicitamente calcular e armazenar uma matriz de pré-condicionamento não é uma tarefa simples. Bons resultados foram obtidos empregando um precondicionador Jacobi, conforme será mostrado mais adiante. Este pré-condicionador representa uma escolha adequada uma vez que é determinado unicamente pelos elementos da diagonal principal da matriz do sistema. Estes *n* elementos podem ser calculados rapidamente quando necessário e o seu armazenamento apenas exigiria um vector auxiliar.

Para um melhor entendimento do método *N-Scheme-GC*, é recomendada uma leitura completa de [19]. Assim, com a modificação do método, uma nova estratégia de paralelização foi aplicada, que é significativamente diferente da anteriormente adotada devido ao número de estruturas de repetições necessárias e da abordagem do método no novo esquema.

Este capítulo apresentou as características dos algoritmos da técnica inicialmente proposta, que será denominada de *N-Scheme-SOR* e sua adaptação com GC, o *N-Scheme-GC*. O capítulo 4 apresentará as técnicas associadas com os conceitos de programação paralela.

3 PROCESSAMENTO PARALELO E DISTRIBUÍDO

Neste capítulo serão apresentadas algumas definições e conceitos básicos, relevantes ao Processamento de Alto Desempenho (PAD) e um breve resumo sobre arquitetura de computadores. Também será apresentado o paradigma de programação paralela em *clusters* de computadores, por troca de mensagens e a utilização de programação paralela *multicore*. Conceitos de *Messaging Passing*, ambiente de programação, rotinas básicas, suporte a biblioteca MPI e programação *multicore* serão também revistos.

3.1 INTRODUÇÃO

Dentre as várias definições para programação paralela existentes na literatura pode-se citar [27] que define a programação paralela como sendo "a atividade de se escrever programas computacionais compostos por múltiplos processos cooperantes, atuando no desempenho de determinada tarefa."

Desde o surgimento da era da computação, a busca crescente por melhores desempenhos computacionais tem motivado a evolução dos computadores, permitindo a implementação e o desenvolvimento de aplicações que permitem uma taxa de computação elevada para pequenos e grandes volumes de dados. Entretanto, para melhorar a capacidade de processamento dos algoritmos, foi necessário aumentar o desempenho dos computadores. Essa necessidade vem de uma época muito mais limitada tecnologicamente e logo, uma das alternativas para superar a defasagem tecnológica da época foi utilizar vários processadores de forma a conseguirem realizar tarefas em paralelo minimizando o tempo do processamento de diversas instruções independentes. Assim, surgiu o termo Processamento Paralelo [28].

As execuções de aplicações mais complexas necessitam de grande poder computacional e exigem também um elevado tempo de resposta. Assim, resolvê-las utilizando a forma de execução sequencial torna essas aplicações pouco eficientes. Para se obter alto desempenho diminuindo esse tempo, é necessário utilizar computadores paralelos, ou seja, dividir uma tarefa em tarefas menores que sejam executadas por máquinas capazes de operar duas ou mais tarefas simultaneamente. Atualmente, já existem disponível no mercado computadores Desktop com arquitetura de processamento paralelo que são classificados como Processamento Paralelo Multicore, ou seja, possuem em um único *Die*

diversos núcleos compartilhando um barramento comum. Existem atualmente equipamentos Desktop e Servidor Multicore com várias capacidades, incluindo 2, 3, 4 e até 8 núcleos em um mesmo chip. Exemplos desses processadores são os Intel Core 2 Duo (multicore com 2 núcleos), Amd Phenom X3 (multicore com 3 núcleos) e com 4 núcleos temos processador da família i7, Amd Phenom, Amd Opteron, Intel Xenon. Outras linhas mais avançadas da Intel e Amd também oferecem produtos com 6 e 8 núcleos, normalmente voltados ao mercado de servidores.

O processamento paralelo não atua apenas na eficiência de um processo em termos de tempo de execução, mas também na busca por uma estruturação melhor e/ou mais segura para o sistema.

Segundo [29], sistemas multiprogramados com um único processador são capazes de processar mais de um programa simultaneamente, compartilhando o tempo de processador entre diversos processos. Entretanto, conforme [30] sistemas multiprocessados compartilham os diversos processadores para executar cada um dos processos existentes, necessitando de um número igual ou superior de processadores em relação ao número de processos para caracterizar um processamento paralelo.

Apesar de todo conceito intrigante e inovador do processamento paralelo, algumas questões merecem atenção antes de usá-lo de forma generalizada. Um destas questões é a revisão de algoritmos tradicionais implementados de forma sequencial. É preciso explorar e reavaliar os princípios do algoritmo para aplicar a nova estrutura de computação e assim obter uma solução paralela que seja eficiente e mais rápida que a solução sequencial.

Sistemas paralelos oferecem maior desempenho para alguns programas lentos, soluções naturais para problemas inerentemente paralelos e uma possível modularidade dos programas. Entretanto, o paradigma de processamento paralelo apresenta algumas dificuldades como a programação propriamente dita, a necessidade de balanceamento de tarefas, bem como a comunicação e o sincronismo entre os processos [31].

Associado a todos esses conceitos e características de um sistema paralelo, está a utilização de sistemas computacionais multicore e distribuídos, permitindo que as tarefas possam ser executadas em um *cluster*, obtendo maior potência computacional, principalmente para aplicações e algoritmos com granularidade alta

3.2 ARQUITETURAS PARALELAS

A busca incessante por modelos de processamento paralelo para conseguir alto desempenho para problemas até então limitados na sua forma sequencial, motivou o surgimento de vários modelos de arquiteturas paralelas. Arquitetura significa estrutura do hardware, capaz de determinar as limitações do uso de uma máquina.

Dentre os modelos de arquitetura paralela existentes, será apresentado o esquema mais utilizado atualmente: a classificação de Flynn [32]. Esta classificação se dá regendo a forma com que os processadores estão conectados à memória e se baseia nos modos de manipulação das instruções e dos dados. Os dados e instruções podem ser únicos ou múltiplos, de acordo com a seguinte simbologia: S – *single*, M – *multiple*, I – *Instruction*, D – *data*.

A classificação de Flynn é amplamente aceita e é a mais utilizada atualmente. Segundo Flynn [32] [33], o processo computacional deve ser visto como uma relação entre fluxos de instruções e fluxos de dados nos processadores, dividindo-se as arquiteturas paralelas em quatro classes conforme ilustra:

- SISD - *single instruction, single data* – computadores com esta característica são aqueles que executam uma instrução de um programa por vez. Ou seja, o modelo tradicional do processador único. Um exemplo seria o computador pessoal com um processador convencional.

- SIMD - *single instruction, multiple data* – neste tipo de arquitetura também tem-se a execução de uma única instrução. Entretanto, devido à existência de facilidades de hardware para armazenamento (vetor ou array), a mesma instrução é processada sob diferentes itens de dados. São normalmente classificadas como: *Processor Arrays* e *Vector Pipeline*. Exemplo de computadores com arquitetura *processor arrays* são as máquinas ILLIAC IV (Universidade de Illinois), Thinking Machine CM-2 e MASPAR MP-1216.
  Exemplo de computadores com arquitetura *vector pipeline* são as máquinas IBM 9000, Cray XM-P, YM-P & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA 10. As GPUs também estão sob esta classificação.

- MISD - *multiple instruction, single data* – um conjunto de dados é colocado concorrente em múltiplas unidades de processamento (UP).

Cada UP opera de maneira independente via conjuntos independentes de instruções.
Algumas utilizações de uma configuração MISD poderiam ser:

- filtros de múltiplas frequências operando um mesmo sinal
- múltiplos algoritmos de criptografia tentando a quebra de uma mensagem codificada.

Em arquitetura de computadores, um array sistólico é considerado uma estrutura MISD [34] [35] [36]. O processador de Rede PXF da Cisco é internamente organizado como um array sistólico [37] e portanto é uma implementação comercial da arquitetura MISD, uma das poucas que se tem conhecimento. Em 1971 uma máquina denominada como C.mmp computer foi desenvolvida na universidade de Carnegie-Mellon.

- MIMD - *multiple instruction, multiple data* – possuem múltiplos processadores cada qual podendo executar diferentes instruções independente dos demais. Estas representam a grande parte das máquinas paralelas atuais e oferecem grande flexibilidade para a construção de algoritmos paralelos. Apesar de independentes, os diversos processadores devem cooperar entre si, tornando necessárias as comunicações e sincronismo entre os processos.

Há uma grande variedade de sistemas MIMD, que são classificados de acordo com os aspectos arquiteturais relevantes no desempenho do sistema como o mecanismo de comunicação/sincronização e a estrutura da memória como mostra a Figura 6.

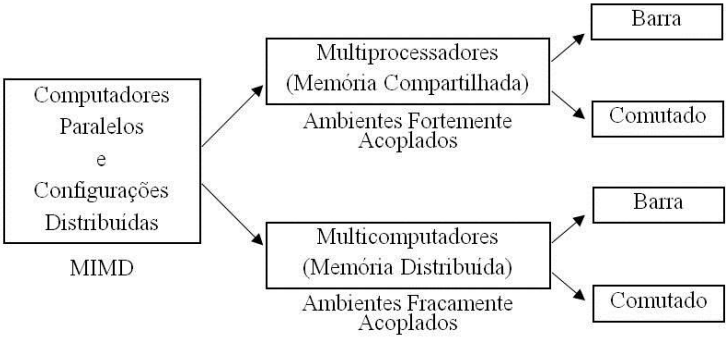

Figura 6 – Taxionomia de arquitetura de computadores MIMD.

3.2.1 Estrutura da Memória

3.2.1.1 Multiprocessadores

São máquinas MIMD que possuem um único espaço para endereçamento de memória, possibilitando regiões de compartilhamento de dados. A memória consiste de vários módulos, de número não necessariamente igual ao número de processadores do sistema. A comunicação e a sincronização entre os processos em execução são feitas através de variáveis compartilhadas, acessíveis por todos eles, o que facilita a programação destes sistemas [38]. A vantagem de um sistema com memória compartilhada é que a comunicação de dados entre os processadores é rápida. A desvantagem desta configuração é que diferentes processadores podem acessar a memória simultaneamente e, nestes casos, haverá um atraso (tempo de contenção) até que a memória esteja livre. A Figura 7 mostra a organização básica de um multiprocessador com sua memória compartilhada [39].

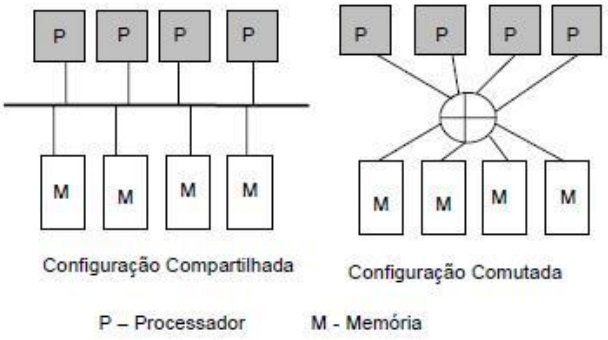

Figura 7 – Multiprocessadores (memória compartilhada).

As máquinas MIMD multiprocessadas apresentam uma rede de interconexão de acesso à memória que pode ser de barramento ou comutada, como mostra a Figura 8.

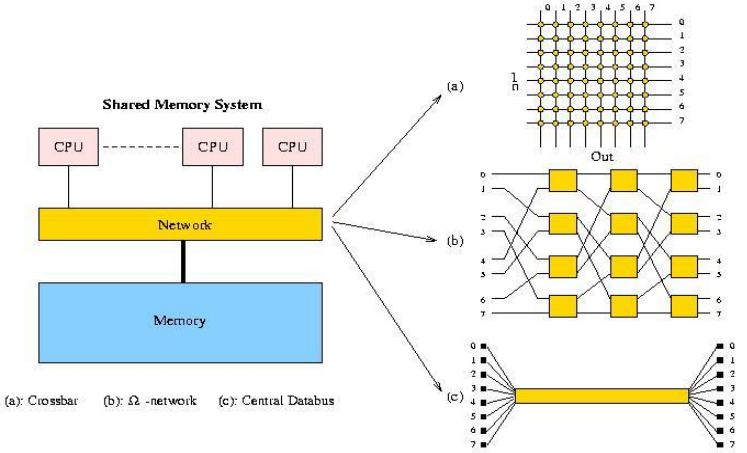

Figura 8 – Redes de interconexão utilizadas em sistemas MIMD com memória compartilhada: Comutada (a) e (b) ou Barramento (c).

A comunicação entre os processadores é feita através de passagem de mensagens, na qual as informações são trocadas entre os processadores. Exemplos destas diferentes formas que cada processador acessa a memória são:

1) Multiprocessadores UMA (*Uniform Memory Access*): esta abordagem é caracterizada por ter todos os elementos processadores com um acesso à memória. Isto significa que todos têm a mesma taxa de transmissão e retardo. Exemplo de uma máquina com esta arquitetura é mostrada na Figura 9:

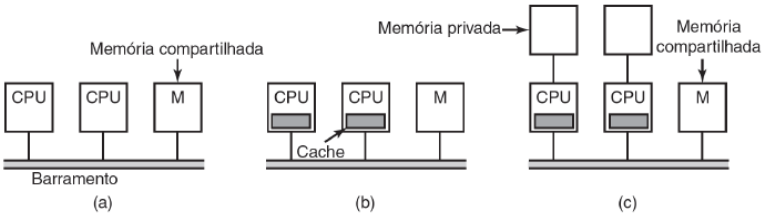

Figura 9 – Multiprocessadores UMA baseados em barramento: (a) sem a utilização de cache, (b) com a utilização de *cache* e (c) com memória privada e utilização de *cache*.

2) Multiprocessadores NUMA (*Non-Uniform Memory Access*): é conhecida por sua característica de poder escalar até centenas de processadores. Existe um espaço de endereçamento único, visível a todos os processadores. O acesso à memória é feita via instrução LOAD e STORE. O acesso a memória local é mais rápido quando comparado ao acesso a uma memória remota.

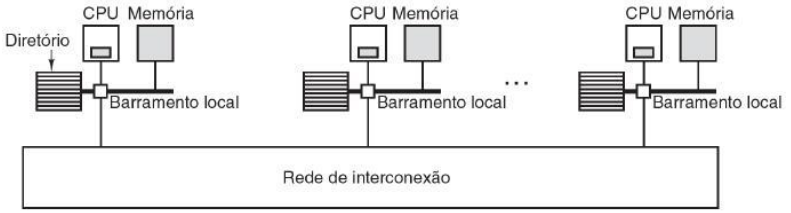

Figura 10 – Multiprocessadores NUMA.

2) Multiprocessadores Simétricos, SMP (*Symmetric MultiProcessor*): ambientes denominados como multiprocessadores simétricos são conhecidos como arquiteturas de compartilhamento total. Estas configurações são caracterizadas por até dezenas de processadores compartilhando todos os recursos computacionais disponíveis e executando um único sistema operacional. Os processadores são considerados simétricos uma vez que têm os mesmos custos para acesso a memória.

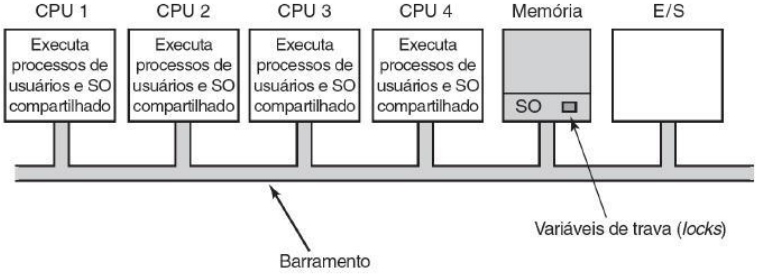

Figura 11 – Multiprocessadores simétricos.

Observa-se na Figura 11 que a configuração é caracterizada por vários processadores compartilhando uma única memória e um único sistema de entrada e saída. Um fator particular da configuração é não possuir múltiplas memórias e nem múltiplos sistemas de entrada e saída, mas apenas múltiplos processadores.

3.2.1.2 Multicomputadores

São máquinas MIMD com vários espaços de endereçamento de memória, não possibilitando memória compartilhada diretamente. Cada conjunto de processador e memória está localizado em computadores fisicamente distintos. A comunicação e a sincronização entre os processos em execução é feita por troca de mensagens, através da rede de interconexão que interliga todos os computadores [38]. A Figura 12 mostra um sistema de memória distribuída [39].

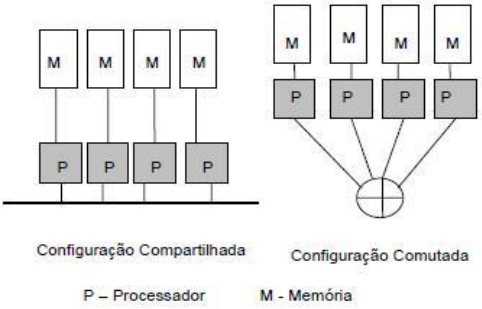

Figura 12 – Multicomputadores (memória distribuída).

As máquinas MIMD multicomputadores apresentam uma rede de interconexão de acesso a memória que pode ser de barramento, Figura 13, ou comutada, Figura 14.

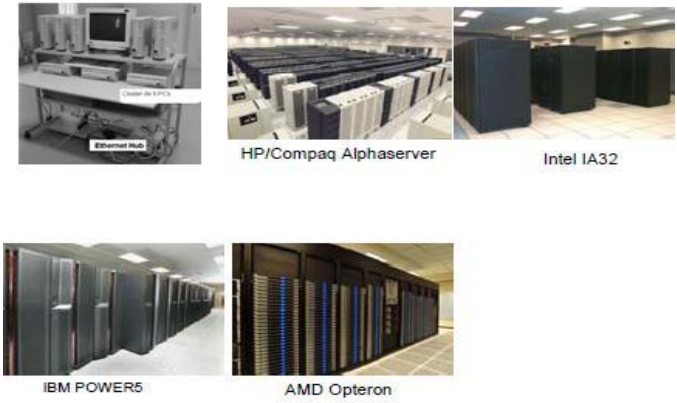

Figura 13 – Redes de interconexão utilizadas em máquinas com memória distribuída (barramento).

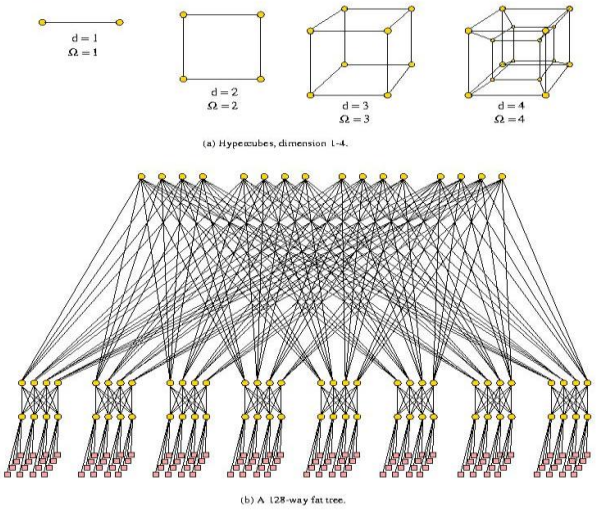

Figura 14 – Redes de interconexão utilizadas em máquinas com memória distribuída (comutada).

As máquinas com configuração MPP (*Massively Parallel Processors*) são usualmente classificadas como multicomputadores. São caracterizados por milhares de nós interligados por dispositivos de conexão de alta velocidade. Cada nó pode ser composto por um ou mais processadores possuindo *cache* e memória locais. Outra característica é que cada nó possui sua própria cópia de sistema operacional, onde as aplicações se executam localmente e se comunicam através de pacotes de troca de mensagens como: MPI – *Message Passing Interface* ou PVM – *Parallel Virtual Machine*.

Arquiteturas de memória centralizada caracterizam-se pela existência de uma memória global e única, a qual é utilizada por todos os processadores, fortemente acoplados, de maneira que através do compartilhamento de posições desta memória ocorre a comunicação entre os processos. Para evitar que a memória se torne um "gargalo", as arquiteturas de memória centralizada devem também implementar mecanismos de memória *cache* [31], uma maneira de reduzir o acesso a memória comum (memória *RAM*).

Nas arquiteturas de memória distribuída, cada processador possui sua própria memória local, sendo então fracamente acoplado. Por não existir compartilhamento de memória, os processos comunicam-se

através de troca de mensagens, que é a transferência explícita de dados entre os processos [31].

Os sistemas distribuídos, sob o aspecto de arquitetura de máquinas para execução de aplicativos, devem ser vistos como configurações com grande poder de escala pela agregação dos computadores existentes nas redes convencionais. Nos ambientes distribuídos, a homogeneidade ou heterogeneidade de um conjunto de máquinas, onde cada qual possui sua arquitetura de software e hardware executando sua própria cópia de sistema operacional, permite a formação de interessantes configurações SMPs, de MPPs, de *clusters* e *grids* computacionais. Dá-se destaque aqui aos agregados de computadores, os multicomputadores, também conhecidos como c*lusters* e os sistemas multiprocessados.

3.3 REDES DE INTERCONEXÃO

Redes de Interconexão (*Interconnection Networks*) são redes de altíssima taxa de transferência projetadas para interconectar processadores e memórias numa arquitetura paralela.

A taxa de transmissão de um canal ou meio físico é a quantidade de *bits* que esse meio consegue transmitir por segundo. Esta taxa pode ser expressa em *bits* por segundo - *bps* (bits per second) - ou kilobits, Megabits ou Gigabits por segundo. As taxas de transmissão entre dois computadores dependem de vários fatores como:

- as características dos cabos utilizados;
- a quantidade de tráfego de mensagens provenientes dos vários nós da rede;
- a utilização de largura de banda para transmissão de um ou vários fluxos de mensagens ao mesmo tempo (multiplexação);
- as taxas máximas de transmissão dos modems ou outros dispositivos de comunicação; etc.

A largura de banda de um cabo ou canal de transmissão de dados é a diferença ou amplitude entre as frequências mais alta e mais baixa que esse canal permite ou utiliza. As frequências são expressas em *hertz*, ou seja, número de ciclos ou impulsos por segundo. A uma maior largura de banda de um canal de transmissão corresponderá uma maior capacidade de transmissão de informação. Essa maior capacidade de transmissão pode se traduzir em taxas de transmissão mais elevada ou

na possibilidade de ser desdobrada em vários fluxos de mensagens ao mesmo tempo (multiplexação).

Multiplexagem - consiste na operação de transmitir várias comunicações diferentes ao mesmo tempo através de um único canal físico. O dispositivo que efetua este tipo de operação chama-se multiplexador.

As métricas utilizadas para comparação das redes de interconexão são:

- Conectividade: nodos de uma rede e os enlaces podem apresentar falhas e devem ser removidos para reparo. A rede deve continuar funcionando com sua capacidade reduzida. O parâmentro conectividade indica a capacidade de flexibilidade da rede continuar a funcionar sob essas condições. A conectividade pode ser entendida como o número mínimo de redes e nodos que, ao falharem, dividem a rede em redes disjuntas. Quanto maior a conectividade, melhor a capacidade da rede em tratar com falhas.

- Diâmetro: representa a distância máxima inter-nodo, ou seja, número máximo de enlaces que devem ser percorridos para envio de uma mensagem para qualquer nodo ao longo do menor caminho. Quanto menor for o diâmetro, menor deverá ser o tempo de envio de uma mensagem de um nodo para outro mais distante.

- Limitação: é a medida de congestionamento na rede.

- Expansão: uma rede deve poder se expandir, criando maiores e mais potentes multicomputadores, através do acréscimo de mais nodos. É desejável que o crescimento seja possível por pequenos incrementos.

Redes de interconexão podem ser configuradas de uma forma estática ou dinâmica. Em outras palavras, totalmente interligada ou interligada dinamicamente. Topologias adotadas e geralmente usadas: *linear array*, *ring*, *star*, *tree*, *nearest neighbor*, *mesh*, *systolic array*, *fully connected*, *3-cube*, *4-cube*. Algumas podem ser observadas na Figura 15.

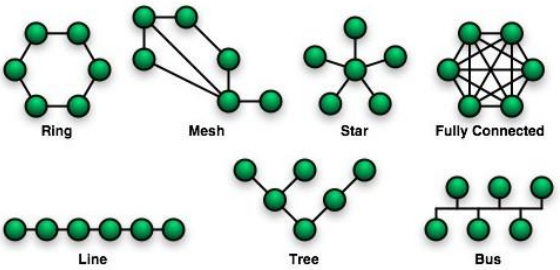

Figura 15 – Algumas topologias de redes de interconexão.

As principais tecnologias de rede utilizadas em arquiteturas paralelas são:

a) *Gigabit Ethernet* – extensões dos padrões 10 *Megabits/s Ethernet* e 100 *Megabits/s Fast Ethernet* para interconexão em redes. Surgiu da necessidade criada pelo aumento da largura de banda nas "pontas" das redes (servidores e estações de trabalho) e com baixo custo.
b) *Myrinet* – desenvolvida pela Myricom, portas e interfaces full-duplex alcançando 1.28 *Gbits/s* para cada link. Controle de fluxo, de erro e monitoramento contínuo dos links. Baixa latência, *switches crossbar* com monitoramento para aplicação de alta disponibilidade. Suporte a qualquer configuração de topologia e com latência de 13 a 21 μs (micro segundos).
c) *Infiniband* – surgiu devido à necessidade de se melhorar o desempenho dos dispositivos de entrada e saída e das comunicações, que surgiu juntamente com o aumento da capacidade de processamento dos processadores. Utiliza uma capacidade hierárquica, com comunicação ponto-a-ponto. Nessa abordagem, todo nó pode ser o iniciador de um canal para qualquer outro. As vantagens são a baixa latência e largura de banda (1,3 μs e 2,5 *Gbits/s*).

3.4 ARQUITETURA DE COMPUTADORES EM *CLUSTERS*

Para suprir a demanda por computação de alto desempenho surge a agregação de microcomputadores os *clusters*, construído a partir de máquinas encontradas no mercado e com tecnologia de rede local (LAN – *Local Area Network*). Os *clusters* referem-se a dois ou mais computadores conectados, podendo estar em único local ou podendo

estar fisicamente separados e conectados via uma LAN. O *cluster* é um tipo de sistema paralelo com arquitetura essencialmente de memória distribuída. Necessitam de uma infraestrutura de comunicação para realizar a conexão entre as máquinas como protocolos que são utilizados para o transporte e fluxo de dados e comunicação em grupo. Esta arquitetura é responsável pela inicialização do programa paralelo para ser executado em múltiplos processadores e provê a utilização de uma biblioteca de funções que permite o envio de dados entre os processadores [40].

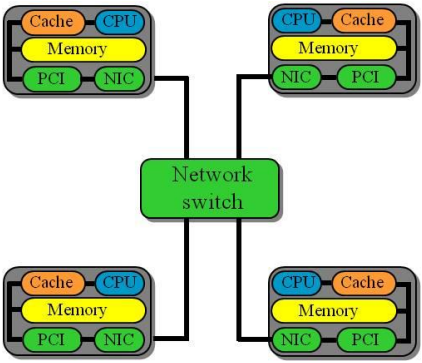

Figura 16 – Arquitetura *cluster* com 4 nós.

Tanto a MPP como a computação em c*luster*, têm uma arquitetura com memória distribuída. A diferença está na sofisticação da interconexão entre os processadores. É a alta largura de banda e a baixa latência do sistema MPP que permite a escalabilidade de milhares de processadores. A baixa escalabilidade dos *clusters* é limitada pela sua rede de interconexão. Portanto, o custo de comunicação em um *cluster* pode se tornar um ponto crítico de perda de desempenho.

3.4.1 *O Ambiente SHARCNET*

A *Sharcnet* (*Shared Hierarchical Academic Research Computing Network*) é um consórcio de instituições acadêmicas canadenses que compartilham uma rede de computadores de alto desempenho [41]. Em parceria com o curso de Ciências da Computação da Universidade Federal de Santa Catarina, foi possível adquirir junto a *Sharcnet* uma conta neste ambiente, onde se torna possível a utilização desta rede de

computadores para realizar os testes necessários para este trabalho de tese.

O gerenciamento do *cluster* é feito remotamente utilizando serviços de rede como FTP (*File Transfer Protocol*) programa para transferência de arquivos, sendo utilizado o SSH (*secure Shell*). Um protocolo de rede e um programa de computador, respectivamente, que permitem a conexão com outro computador na rede, de forma a executar comandos na unidade remota. As configurações, programas e arquivos são armazenados no *cluster* servidor e quando os nós (máquinas) precisam de acesso a alguns arquivos, isto é feito através da conexão de rede para leitura/escrita de arquivos.

As características dos sistemas que podem ser utilizados como ambiente de pesquisa e experimentação para os testes realizados neste trabalho de tese podem ser visualizados com mais detalhes em [41]. Uma apresentação geral dos sistemas existentes na *Sharcnet* pode ser visualizada na Figura 17, Figura 18 e Figura 19. Na Figura 20 tem-se a área do sistema de rede do ambiente da *Sharcnet*.

Core Systems

System	State	Cores	Architecture	Nodes	Node Memory	Notices
kraken	Online	3760	Cluster/Myrinet 2g (gm)	Opteron	4,8,32 GB	09-Aug-2011
orca	Online	7680	Cluster/QDR InfiniBand	Opteron	32 GB	22-Jul-2011
requin	Online	1536	Cluster/Quadrics Elan4	Opteron	4,8 GB	05-Aug-2011
saw	Online	2688	Cluster/DDR InfiniBand	Xeon	16 GB	28-Jul-2011

Figura 17 – *Core systems* para aplicações paralelas com MPI.

Specialty Systems

System	State	Cores	Architecture	Nodes	Node Memory	Notices
angel	Online	176	Accelerator/InfiniBand	Xeon/gpu	8,16 GB	22-Jul-2011
gulf	Online	0	Storage/GigE	N/A	N/A	27-Jul-2011
hound	Online	480	Cluster/InfiniBand	Xeon, Opteron	8,128 GB	04-Aug-2011
lundun	Online	0	Storage/GigE	N/A	N/A	27-Jul-2011
mako	Online	240	Cluster/GigE	Xeon/Nehalem	2,8,16 GB	06-Jun-2011
megamouth	Online	0	Storage/GigE	N/A	N/A	28-Jun-2011
prickly	Online	40	Accelerator/Gigbit Ethernet	IBM Cell	8,16 GB	21-Jun-2011
rainbow	Online	80	Cluster/InfiniBand	Opteron	8 GB	21-Jul-2011
silky	Offline	128	SMP/NUMA	Itanium2	256 GB	10-Aug-2011
tope	Online	8	SMP/Internal	Xeon	32 GB	04-May-2011

Figura 18 – *Specialty systems* para aplicações paralelas com GPU (*graphics processing unit*) e aplicações *multithread*.

Cada um destes sistemas possui diferentes arquiteturas, tipos de interconexão de rede e diferentes números de processadores. Tais sistemas têm softwares disponíveis destinados a vários tipos de aplicações. Do grupo de sistemas acima, utilizou-se um dos *clusters* do *Core Systems* para as execuções dos problemas testes apresentados neste trabalho.

Contributed Systems

System	State	Cores	Architecture	Nodes	Node Memory	Notices
bramble	Online	64	SMP/NUMA	Itanium2	128 GB	15-Apr-2011
brown	Online	960	Cluster/Gigabit Ethernet	Xeon	12,16,32,48 GB	25-Jul-2011
goblin	Online	324	Cluster/Gigabit Ethernet	Opteron	4,8,12,48 GB	08-Jul-2011
gulper	Online	82	Cluster/Myrinet 2g (mx)	Opteron	2,4,24 GB	27-Apr-2011
guppy	Online	272	Cluster/QDR InfiniBand	Xeon	24 GB	07-Jun-2011
prism	Online	4	SMP/NUMA	Itanium2	8 GB	27-Sep-2010
redfin	Offline	528	Cluster/Infiniband	Opteron	98 GB	
school	Online	8	SMP/SMP	Itanium2	16 GB	27-Sep-2010
wobbie	Online	360	Cluster/Myrinet 2g (mx)	Opteron	2,4,8,32 GB	21-Jul-2011

Figura 19 – *Contributed systems* para aplicações diversas.

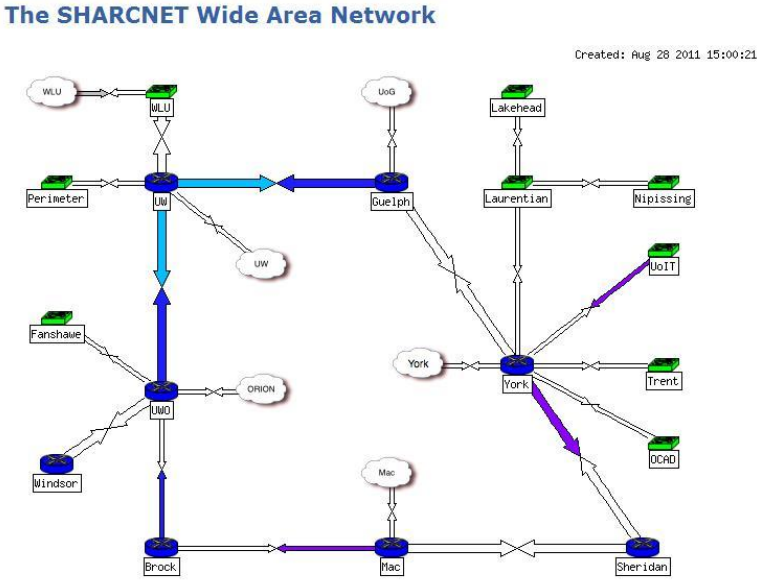

Figura 20 – Área de rede do ambiente da *Sharcnet*.

3.5 A PROGRAMAÇÃO PARALELA

Na busca por um ganho de desempenho na execução de aplicações em paralelo é importante que o algoritmo do problema em questão seja paralelizado de forma explícita (manual), ou implícita (automaticamente) [40].

A exploração implícita do paralelismo é empregada com a utilização de ferramentas automáticas (compiladores paralelizantes). O compilador irá automaticamente detectar os pontos onde é possível uma execução em paralelo e gerar um código em paralelo. Essa técnica é altamente dependente da arquitetura de hardware e de uma ferramenta para paralelização automática para a linguagem de programação utilizada. Contudo, a exploração implícita não pode ser considerada eficaz por algumas razões como: não ocorrer uma paralelização total do código e não serem aplicáveis a qualquer tipo de aplicações e geralmente geram códigos menos eficientes.

Por outro lado, a exploração explícita do paralelismo exige que o programador especifique os pontos onde o paralelismo será explorado e como a tarefa será realizada. Ela possibilita a escrita de programas com vários fluxos de execução isolados que deverão ser executados de forma independente e concomitantemente.

Neste trabalho de tese escolheu-se pela técnica de exploração explícita do paralelismo, visto que o programador possui um controle maior sobre os aspectos de sincronismo e comunicação entre os processos.

Além disso, a programação paralela está associada às arquiteturas paralelas: programação utilizando memória compartilhada e a programação utilizando troca de mensagens. Atualmente, a programação utilizando troca de mensagens é muito utilizada e requer que o programador mova conjunto de dados através da memória por comunicações explícitas. Os processos devem se comunicar através do envio de mensagens, ou seja, mensagens de envio e recebimento de dados.

O nível de granularidade ou paralelismo depende de como acontecem as comunicações entre os processos e isso depende da aplicação em questão e da técnica de programação adotada. A granularidade está diretamente relacionada com o tempo de computação gasto em trocas de informações entre os processadores ou processos. Ou seja, a granularidade do problema refere-se à forma de como uma tarefa é paralelizada.

Na literatura, pode-se encontrar três níveis de granularidade:

- *Grossa*: indica tarefas grandes com pouca comunicação, aplicada em plataformas com pouco nível de interação;
- *Fina*: indica muitas tarefas pequenas com muita comunicação, muito rápida e confiável;
- *Média*: situa-se em um patamar entre as duas anteriores.

3.6 TIPOS DE PARALELISMO

3.6.1 Paralelismo dados (SPMD)

SPMD (*Single Program, Multiple Data*). O processador executa as mesmas instruções sobre dados diferentes, como mostra a Figura 21. É aplicado, por exemplo, em programas que utilizam grandes matrizes e para cálculos de elementos finitos [42].

Figura 21 – Paralelismo de dados.

3.6.2 Paralelismo funcional (MPMD)

O processador executa instruções diferentes que podem ou não operar sobre o mesmo conjunto de dados, como mostra a Figura 22. Este é aplicado em programas dinâmicos e modulares onde cada tarefa será um programa diferente [42].

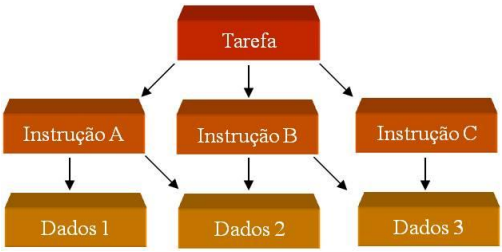

Figura 22 – Paralelismo funcional.

Os modelos fundamentais de programação paralela são os modelos mestre-escravo e o modelo SPMD. No modelo mestre-escravo, as tarefas executadas pelo processo mestre são diferentes das tarefas executadas pelos processos escravos. O processo mestre centraliza o controle, inicializando os escravos e gerenciando o envio de tarefas para os mesmos. O mestre também é responsável pelo recebimento dos resultados calculados pelos escravos, assim como, a finalização dos mesmos após o término do processamento. Em outras palavras, distribui as tarefas, sincronizando a comunicação e executando as operações de I/O. O processo mestre pode ou não contribuir no processamento de tarefas, mas geralmente os escravos executam todo o processamento.

No modelo SPMD (*Single Program, Multiple Data)* não existe um controle centralizado e todos os processadores executam o mesmo programa, contribuindo para o processamento. Para tanto, neste tipo de modelo é necessário que os cálculos e as comunicações entre os processadores sejam coordenados em determinados pontos.

A aplicação segue um modelo MPMD (*multiple program, multiple data*) quando duas ou mais sequências de instruções distintas e independentes são executadas sobre diferentes fluxos de dados, e se adapta a máquinas paralelas do tipo MIMD (vide Figura 6).

Parte deste trabalho foi desenvolvida para executar em *clusters* de computadores, ou seja, realizado em uma arquitetura de memória distribuída. O modelo de programação utilizado foi o SPMD (quanto à localização dos dados e do programa executável) e o modelo mestre-escravo (quanto à divisão de tarefas). Para isso, foi utilizado o paradigma de troca de mensagens através da biblioteca MPI (*Message Passing Interface*) [43]. A outra parte foi desenvolvida para executar em ambientes multiprocessados, em arquiteturas *multicore* através da programação com *Threads*. As características dos modelos de programação paralela serão apresentadas nas seções, 3.8 e 3.9 respectivamente.

3.7 PROCESSAMENTO DE ALTO DESEMPENHO (PAD)

Uma definição mais restrita e que ao mesmo tempo ilustre como as técnicas de alto desempenho podem ser aplicadas, pode ser [13]:

“Computação de alto desempenho agrupa uma coleção de sistemas de hardware, ferramentas de software, linguagens e abordagens genéricas de programação, que tornam possíveis, a um preço apropriado, aplicações antes infactíveis.”

A PAD depende de fatores como [13]:

- O tipo de compilador utilizado para a geração do código;
- Otimizações realizadas nas declarações de linhas de código;
- A aplicação de autoparalelização realizada pelo próprio compilador;
- A utilização de bibliotecas de comunicação;
- Medidas de tempo e de recursos de hardware disponíveis.

A geração de códigos mais rápidos depende de cada compilador, já que cada um tem um conjunto diferente de funcionalidades. As otimizações nas linhas de código incluem o reordenamento de laços iterativos no acesso de vetores; a redução do esforço em chamadas a funções específicas como utilização de produto ao invés de uma exponencial; a remoção de código sem efeito como funções que nunca são chamadas e variáveis que são calculadas, mas não são usadas novamente, o reordenamento na declaração de variáveis e dezenas de outros cuidados.

PAD envolve alguns conceitos que são muitos citados junto às especificações de hardware e software. Métricas de desempenho servem para analisar o comportamento de sistemas paralelos. Dentre elas, a de maior relevância é o tempo de execução, sendo importantes também o *speedup*, a sincronização, o *overhead*, a granularidade, a eficiência, a escalabilidade e a latência [31] [38] [39] [40].

O tempo de execução refere-se ao tempo decorrido desde o início até o fim da execução de uma determinada tarefa, seja no processamento sequencial ou no paralelo.

O s*peedup* é a medida que tem por objetivo determinar a relação existente entre o tempo dispensado para executar um algoritmo em um único processador (t_s) e o tempo gasto para executá-lo em '*p*' processadores (t_p):

$$S(p) = \frac{tempo\ de\ execução\ \ sequencial}{tempo\ de\ execução\ paralela} = \frac{t_s}{t_p} \qquad (3.1)$$

Teoricamente, para um programa paralelo executado por '*p*' processadores, cada processador é responsável por *1/p* do processamento, então o s*peedup* máximo será '*p*'. Quando o *speedup* se iguala a '*p*' indica que o caso ótimo foi atingido. Entretanto, fatores

como sincronizações e o*verheads* são fatores que afetam diretamente o resultado do *speedup*.

A *eficiência*, por sua vez, é a razão do *speedup* em relação ao número de processadores ou processos utilizados. Denota a qualidade da implementação paralela do algoritmo para o tamanho de problema em questão.

$$E(p) = \frac{S(p)}{p} \tag{3.2}$$

Sincronização é a coordenação temporal de tarefas, que implica a espera pelos processos da finalização de duas ou mais tarefas em um determinado ponto da linha de código (ponto de sincronismo).

Overhead é um desperdício de tempo que ocorre durante a execução dos processos. Os fatores que levam ao *overhead* são: comunicação, tempo em que a máquina fica ociosa e computação extra. Em outras palavras, é o tempo utilizado para coordenar tarefas, transferir dados entre processadores, entre outros consumos de processamento que não são encontrados em programas sequencias.

Granularidade é uma medida da razão entre a quantidade de computação realizada em uma tarefa paralela e a quantidade de comunicação necessária. Esta grandeza determina se um algoritmo pode ou não ser utilizado na arquitetura paralela pretendida.

Entende-se por *escalabilidade* de um sistema paralelo sua capacidade de manter um *speedup* crescente proporcionalmente ao número de processadores utilizados. Em outras palavras, permitir que um sistema possa ter seu tamanho aumentado tal que o seu desempenho também aumente, significa escalabilidade do hardware ou do software.

Latência é também conhecida como atraso, representa a expressão do tempo necessário para um pacote de dados ir de um ponto para outro. Em outras palavras, é a referência a qualquer atraso ou espera que aumente o tempo de resposta real além do tempo de resposta desejado.

3.8 PROGRAMAÇÃO POR TROCA DE MENSAGENS

3.8.1 A Biblioteca MPI

A biblioteca MPI é resultado de um esforço da comunidade para definir e padronizar a sintaxe e semântica de uma biblioteca de rotinas para troca de mensagens que pudesse ser implementada em uma ampla

variedade de máquinas paralelas. É uma das bibliotecas mais difundidas, sendo padrão para a comunicação por troca de mensagens em agregados de computadores (*clusters*). Esta ainda pode ser utilizada em máquinas com memória distribuída, com memória compartilhada, redes comuns e até mesmo em um único processador para emular paralelização [28].

Esta biblioteca possui aproximadamente 125 funções para programação e ferramentas para se analisar o desempenho utilizando programas em C, C++ e FORTRAN.

As bibliotecas de *Message Passing* possuem rotinas com finalidades bem específicas como [42]:

- Rotinas de gerência de processos:
  - Inicializar e finalizar processos
  - Determinar número de processos
  - Identificar processos
- Rotinas de comunicação ponto-a-ponto:
  - Comunicação síncrona ou assíncrona
  - Bloqueante ou não-bloqueante
  - Empacotamento de dados
- Rotinas de comunicação por grupos:
  - Broadcast
  - Sincronização de processos (barreiras)

3.8.2 Implementações

As duas principais bibliotecas de *Message Passing* são [42]:

- PVM – (Parallel Virtual Machine) - Possui como característica o conceito de "máquina virtual paralela", dentro da qual processadores recebem e enviam mensagens, com finalidades de obter um processamento global;
- MPI – Conhecido como "*message passing interface*", que está se tornando um padrão na indústria.

As principais implementações de MPI são:

- IBM MPI - Implementação IBM para SP e clusters;
- MPICH - Argonne National Laboratory/Mississipi State University;
- UNIFY - Mississipi State University;
- CHIMP - Edinburgh Parallel Computing Center;
- LAM - Ohio Supercomputer Center;

- PMPIO - NASA;
- MPIX - Mississippi State University NSF Engineering Research Center.

3.8.3 Conceitos Básicos de MPI

***Processo*:** Por definição, cada programa em execução constitui um processo. Considerando um ambiente multiprocessado, pode-se ter processos em inúmeros processadores.

Mensagem (message): É o conteúdo de uma comunicação, formado de duas partes:

- **Envelope** = Endereço (origem ou destino) e rota dos dados. O envelope é composto de três parâmetros: identificação dos processos (transmissor e receptor); rótulo da mensagem; comunicador.
- **Dado** = Informação que se deseja enviar ou receber. É representado por três argumentos: endereço onde o dado se localiza; número de elementos do dado na mensagem; tipo do dado.

Rank - Todo processo tem uma única identificação, atribuída pelo sistema quando o processo é inicializado. Essa identificação é contínua, representada por um número inteiro, começando em zero até *p-1* processos, onde *p* é o número de processos.

Group - Grupo é um conjunto de *p* processos. Todo e qualquer grupo é associado a um comunicador e, inicialmente, todos os processos são membros de um grupo com um comunicador preestabelecido (MPI_COMM_WORLD).

Communicator - O comunicador define um conjunto de processos que poderão se comunicar entre si. O MPI utiliza essa combinação e o contexto para garantir uma comunicação segura e evitar problemas no envio de mensagens entre os processos. A maioria das rotinas do MPI exige que seja especificado um comunicador como argumento: O MPI_COMM_WORLD é o comunicador pré-definido que inclui todos os processos definidos pelo usuário em uma aplicação MPI. [43]

Application Buffer - É o endereço de memória, gerenciado pela aplicação, que armazena um dado que o processo necessita enviar ou receber.

System Buffer **-** É um endereço de memória reservado pelo sistema para armazenar mensagens. Dependendo do tipo de operação de envio e/ou recebimento (*send e/ou receive*), o dado no buffer da aplicação (*application buffer*) pode necessitar ser copiado de e/ou para o buffer do sistema (system buffer) através das rotinas *send buffer e/ou receive buffer*. Neste caso a comunicação é assíncrona.

Blocking Comunication **-** Em uma rotina de comunicação blocking, a finalização da chamada depende de certos eventos. Em uma rotina de envio de mensagem, o dado tem que ter sido enviado com sucesso, ou ter sido salvo no buffer do sistema (*system buffer*), indicando que o endereço do buffer da aplicação (*application buffer)* pode ser reutilizado. Em uma rotina de recebimento de mensagem, o dado tem que ser armazenado no buffer do sistema (*system buffer*), indicando que o dado pode ser utilizado.

Non-Blocking Comunication **-** Em uma rotina de comunicação non-blocking, a finalização da chamada não espera qualquer evento que indique o fim ou sucesso da rotina. As rotinas non-blocking comunication não esperam pela cópia de mensagens do buffer da aplicação (*application buffer*) para o buffer do sistema (*system buffer*), ou a indicação do recebimento de uma mensagem.

3.8.4 Técnicas de comunicação entre os processos

Buffering - É a cópia temporária de mensagens entre endereços de memória efetuada pelo sistema como parte de seu protocolo de comunicação. A cópia ocorre entre o buffer do usuário (*application buffer*) definido pelo processo e o buffer do sistema (*system buffer*) definido pela biblioteca.

Utilização: Muitas vezes, pode ser necessário utilizar explicitamente um novo local para armazenar os dados. Um exemplo disso ocorre quando se pretende enviar uma grande massa de dados.

Blocking **-** A rotina de comunicação é blocking quando a finalização da execução da rotina é dependente de determinados eventos, ou seja, espera por determinada ação antes de liberar a continuação do processamento.

*Utilização***:** Quando se deseja obter confirmação de recebimento de determinados dados para continuar o processamento, sempre que se criam "barreiras" de controle entre os processos.

Non-Blocking - A rotina de comunicação é non-blocking quando a finalização da execução da rotina não depende de determinados eventos, ou seja, o processo continua sendo executado normalmente sem haver espera.

Utilização: Quando não existem dependências ou necessidade de controle intensivo.

Assíncrono - É a comunicação na qual o processo que envia a mensagem não espera que haja um sinal de recebimento da mensagem pelo destinatário.

Utilização: Em comunicação confiável, principalmente.

Síncrono - É a comunicação na qual o processo que envia a mensagem não retorna a execução normal enquanto não haja um sinal do recebimento da mensagem pelo destinatário.

Utilização: Quando, por questões de segurança (como por exemplo, devido às instabilidades da rede), se torna necessária uma confirmação da comunicação.

3.8.5 Tipos de comunicação no MPI

Existem os seguintes tipos de comunicação entre os processos detalhados a seguir [42] [43].

3.8.5.1 Ponto-a-Ponto

Dentre as rotinas básicas do MPI estão as rotinas de comunicação ponto-a-ponto que executam a transferência de dados entre dois processos.

Existem quatro (4) modos de comunicação ponto a ponto:

Synchronous (Síncrono) - É a comunicação na qual o processo que envia a mensagem não retorna a execução normal enquanto não haja um sinal do recebimento da mensagem pelo destinatário.

Ready (Imediata) - Neste tipo de comunicação sabe-se, a priori, que o recebimento (*receive*) de uma mensagem já foi efetuado pelo processo destino, podendo-se enviá-la (*send*) sem necessidade de confirmação ou sincronização, melhorando o desempenho na transmissão.

Buffered (Buferizada) - No modo buferizada, a operação de envio (*send*) utiliza uma quantidade de espaço específica para o buffer definida pelo usuário (*application buffer*). Isto se tornará necessário

quando a quantidade de mensagem a ser enviada ultrapassar o tamanho padrão do buffer de 4Kbytes para um processo.

Standard (Padrão) - É um modo padrão para o envio (*send*) de mensagens, buscando um meio termo entre eficiência e segurança, usando comunicação *blocking*.

Além dos tipos de comunicação citados acima, temos duas formas de chamadas a rotinas:

Blocking (Blocante) - A rotina de comunicação é blocking quando a finalização da execução da rotina é dependente de determinados eventos, ou seja, espera por determinada ação antes de liberar a continuação do processamento.

Non-Blocking (Não-Blocante) - A rotina de comunicação é non-blocking quando a finalização da execução da rotina não depende de determinados eventos, ou seja, o processo continua sendo executado normalmente sem haver espera.

3.8.5.2 Coletiva

É a comunicação padrão que invoca todos os processos em um grupo (*group*). Normalmente a comunicação coletiva envolve mais de dois processos. As rotinas de comunicação coletiva são voltadas para a comunicação e/ou coordenação de grupos de processos. Existem os seguintes tipos de comunicação coletiva:

Broadcast (Difusão) - É a comunicação coletiva em que um único processo envia (*send*) os mesmos dados para todos os processos com o mesmo communicator.

Reduction (Redução) - É a comunicação coletiva onde cada processo no communicator contém um operador, e todos eles são combinados usando um operador binário que será aplicado sucessivamente.

Gather (Coleta) - A estrutura dos dados distribuídos é coletada por um único processo.

Scatter (Espalhamento) - A estrutura dos dados que está armazenada em um único processo é distribuída a todos os processos.

3.8.6 Rotinas básicas do MPI

Para um grande número de aplicações, um conjunto de apenas seis sub-rotinas MPI é suficiente para desenvolver uma aplicação em paralelo. A seguir, as sintaxes destas rotinas na linguagem FORTRAN 77 e C++, respectivamente:

✓ **MPI_INIT** – inicializar um processo

Para a inicialização de um processo MPI, a primeira rotina a ser chamada deve ser o MPI_INIT que estabelece o ambiente necessário para a execução do MPI. A rotina tem a finalidade de sincronização de todos os processos na inicialização de uma aplicação MPI.

Sintaxe:

MPI_INIT (mpierr)

MPI_Init (&argc, &argv)

Onde: mpierr ou argc/argv – variável inteira de retorno com o status da rotina.

✓ **MPI_COMM_RANK** – identificar um processo

Esta rotina identifica um processo dentro de um grupo de processos e retorna um valor inteiro entre 0 e *p-1* processos.

Sintaxe:

MPI_COMM_RANK (MPI_COMM_WORD, rank, mpierr)

MPI_Comm_rank (MPI_COMM_WORD, &rank)

Onde: MPI_COMM_WORD (default) – comunicador MPI;

rank – variável inteira de retorno com o número de identificação do processo;

✓ **MPI_COMM_SIZE** – contagem de processos

Esta rotina retorna o número de processos dentro de um grupo.

Sintaxe:

MPI_COMM_SIZE (MPI_COMM_WORD, size, mpierr)

MPI_Comm_size (MPI_COMM_WORLD, &size)

Onde: size – variável inteira que retorna o número de processos inicializados durante uma aplicação MPI.

✓ **MPI_SEND** – envio de mensagens

Rotina de envio de mensagens blocante (*blocking send*). Só retorna quando a mensagem for recebida pelo destinatário. A mensagem está salva e o *buffer* do sistema pode ser reutilizado.

Sintaxe:

MPI_SEND (buf, count, datatype, dest, tag, comm, mpierr)
MPI_Send (&buf, count, datatype, dest, tag, comm)
Onde: buf – contém o endereço dos dados a serem enviados;
count – número de elementos a serem enviados;
datatype – tipo de dado;
dest – identificação do processo destino;
tag – rótulo da mensagem;
comm – MPI_COMM_WORLD.

✓ **MPI_RECV** – recebimento de mensagens
Rotina de recebimento de mensagens blocante.
Sintaxe:
MPI_RECV (buf, count, datatype, source, tag, comm, status, mpierr)
MPI_Recv (&buf, count, datatype, source, tag, comm, &status)

Onde: buf – contém o endereço para onde os dados serão enviados;
source – identificação do processo de origem;
status – vetor com informações de source.

✓ **MPI_FINALIZE** – finaliza o processo
Esta é a última rotina executada por uma aplicação MPI, sincronizando todos os processos na finalização da aplicação.
Sintaxe:
MPI_FINALIZE (mpierr)
MPI_Finalize()

Além dessas seis sub-rotinas básicas, serão utilizadas no processo de implementação algumas funções de comunicação como as sub-rotinas:

✓ **MPI_BCAST** – envia dados para todos os processos
Rotina que possibilita enviar/receber dados simultaneamente de/para vários processos.
Sintaxe:
MPI_BCAST (buffer, count, datatype, source, comm, mpierr)
MPI_Bcast(&buffer, count, datatype, source, comm)
Onde: buffer – contém o endereço do dado a ser enviado.

- ✓ **MPI_GATHER** – coleta dados de processos

Rotina que possibilita a coleta de mensagens de um subgrupo de processos.

Sintaxe:

MPI_GATHER (sbuf, scount, datatype, rbuf, rcount, datatype, source, comm, mpierr)

MPI_Gather (&sbuf, scount, datatype, &rbuf, rcount, datatype, source, comm)

Onde: sbuf – contém o endereço inicial do dado a ser coletado;
scount – número de elementos a serem coletados;
rbuf – contém o endereço onde os dados serão armazenados;
rcount – número de elementos recebidos por processo.

A Figura 23 é um exemplo de um programa escrito em FORTRAN com chamadas a rotinas básicas do MPI. Esse programa descreve os comandos de uma aplicação em paralelo com:

- ✓ Biblioteca MPI (linha 2);
- ✓ A inicialização da aplicação MPI (linhas 6 a 8);
- ✓ O envio de uma mensagem "*olá, mundo*" pelo processo denominado de mestre (*rank = 0*), através da rotina MPI_SEND (linhas 10 a 15);
- ✓ O recebimento da mensagem pelos demais processos denominados de escravos (*rank ≠ 0*), através da rotina MPI_RECV (linhas 16 a 19);
- ✓ E a finalização da aplicação MPI (linha 21).

```
1 program hello
2 include "mpif.h"
3 integer mpierr, rank, size, i ,tag
4 integer status(MPI_STATUS_SIZE)
5 character message*11
6 call MPI_INIT(mpierr)
7 call MPI_COMM_RANK(MPI_COMM_WORD,rank mpierr)
8 call MPI_COMM_SIZE(MPI_COMM_WORD,size,mpierr)
9 tag=100
10 if (rank.eq.0) then
11    message = 'Ola, Mundo!'
12    do i=1, size-1
13          MPI_SEND(message,11,MPI_CHARACTER,i,tag,
14&               MPI_COMM_WORD,mpierr)
```

```
15    enddo
16 else
17    MPI_RECV(message,11,MPI_CHARACTER,0,tag,
18&         MPI_COMM_WORD,status, mpierr)
19 endif
20 print*,'Mensagem do no',rank,':',message
21 call MPI_FINALIZE(mpierr)
22 end
```

Figura 23 – Programa exemplo em FORTRAN 77 com a biblioteca MPI [42].

3.9 PROGRAMAÇÃO MULTICORE

A Programação *Multicore* é a programação paralela mais comum e difundida atualmente, isso graças a diminuição do custo dos computadores modernos que, em sua maioria, utilizam arquitetura multicore.

A grande vantagem do desenvolvimento de aplicativos multicore modernos está no fato do controle do fluxo do programa estar destinado ao Sistema Operacional a qual é executado. As implementações de Sistema Operacionais modernos utilizam de um *Scheduler* [44] [45] que é responsável por comutar os processadores e o tempo de processador destinado a cada execução, sendo assim possível, por exemplo, executar um programa multicore mesmo em sistemas *Singlecore* ou com uma quantidade menor de núcleos e ou *threads* virtuais. Isso é possível, decorrente justamente do *Scheduler* do sistema operacional que definirá se há disponibilidade do processador ou se deverá aguardar em fila. Para obter proveito destas facilidades, deve-se utilizar chamadas de sistema, como *fork( )* [45], em sistema que utilizam padrões POSIX ou *CreateProccess( )* no Microsoft Windows [46].

A utilização de chamadas de sistema como *fork( )* realiza um cópia do processo atual através da função *clone( )* retornando a identificação do processo (PID) pai e filho para ser integrado a lógica do algoritmo. Existem também outras funções e bibliotecas capazes de paralelizar a execução do código, muitos destes recursos estão relacionados a criação de *Threads*. *Threads* agem como processos leves e são a menor parte das instruções de um programa que podem ser gerenciados pelo *Scheduler* (escalonador) do Sistema Operacional.

É muito comum confundir *threads* com *processos*, inclusive por oferecem na maioria das vezes resultados similares, porém, em sistemas operacionais, processo é um módulo executável único, que corre

concorrentemente com outros módulos executáveis. Por exemplo, em um ambiente multi-tarefa (como o Unix) que suporta processos, um processador de texto, um navegador e um sistema de banco de dados são processos separados que podem rodar concomitantemente. Processos são módulos separados e carregáveis, ao contrário de *threads*, que não podem ser carregadas. Múltiplas *threads* de execução podem ocorrer dentro de um mesmo processo. Além das *threads*, o processo também inclui certos recursos, como arquivos e alocações dinâmicas de memória e espaços de endereçamento [44] [47].

Programação *multicore* permite acrescentar simultaneidade aos programas escritos em C, C++ e FORTRAN sobre a base do modelo de execução *fork-join* ilustrado na Figura 24. Está disponível em muitas arquiteturas, incluindo as plataformas Unix e Microsoft Windows. É um modelo de programação portável e escalável que proporciona aos programadores uma interface simples e flexível para o desenvolvimento de aplicações paralelas para as plataformas que vão dos computadores de escritório até os supercomputadores [48] [49].

3.9.1 Paralelismo em ambientes *Multicore*

Programação com *multicore* é uma implementação de *multithreading*, um método de paralelização através do uso de *threads*. Um *thread* de execução é a menor unidade de processamento que pode ser programado por um sistema operacional.

Threads existem dentro dos recursos de um único processo. Sem o processo, eles deixam de existir. O número de *threads* a serem executados paralelamente depende do processador, normalmente o número de *threads* que podem ser executados simultaneamente está relacionado a quantidade de núcleos, porém modelos mais modernos de processadores como o processador i7 da Intel permitem 2 *threads* por núcleo atravez da Tecnologia Hyper-threading da Intel [50]. Outros tipos de processadores como a série Niagara Sparc que tem entre 8 a 32 *threads* por núcleo [51]. No entanto, a utilização efectiva das *threads* depende da aplicação.

Todo programa inicia com um único processo denominado *master thread*. Um número específico de *workers threads* são definidos e uma tarefa é dividida entre eles. Os *threads* são então executados simultaneamente em um ambiente de execução, distribuindo as *threads* para diferentes processadores [48].

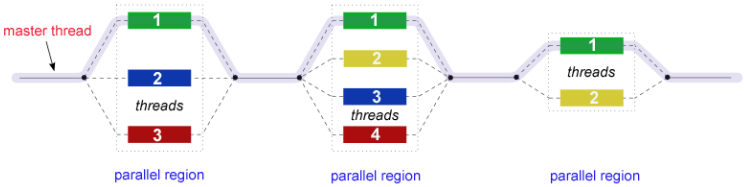

Figura 24 – Modelo *fork-join*.

Fork: o *master thread* cria uma equipe de *threads* paralelas;

Join: quando a equipe de *threads* completa a execução da região paralelizada, eles sincronizam e terminam, voltando a execução para o *master*.

Todo esse procedimento é realizado através de um conjunto de diretivas e rotinas que permitem a criação, gerenciamento e finalização dos *threads*. A parte de código que é criada para funcionar em paralelo é marcada de acordo com uma diretriz pré-processador que criará os *threads* para formar a seção antes de ser executada. Cada *thread* tem um "id" (endereço) anexado a ele e sua criação, em sistemas POSIX, é feita utilizando a biblioteca **pthread.h** com a função **pthread_create( )**. Com a função **pthread_join( )** é possível realizar um *join* após as *threads* completarem a execução [52].

O *thread* "id" é um número inteiro e o *master thread* possui o id "0". Após a execução do código em paralelo, os *threads* retornam ao *master thread*, o qual continua progressivo até o fim do programa [49].

A função *fork( )* é uma função que duplica o processo atual dentro do sistema operacional. O processo que inicialmente chamou a função *fork( )* é chamado de processo pai (*parent*). O novo processo criado pela função *fork( )* é chamado de processo filho (*child*) como mostra a Figura 25. Todas as áreas do processo são duplicadas dentro do sistema operacional (código, dados, pilha, memória dinâmica) [53].

A função *fork( )* é chamada uma única vez (no pai) e retorna duas vezes (uma no processo pai e outra no processo filho). O processo filho herda informações do processo pai:

- Usuários (user id) Real, efetivo;
- Grupos (group id) Real, efetivo;
- Variáveis de ambiente;
- Descritores de arquivos;
- Prioridade;

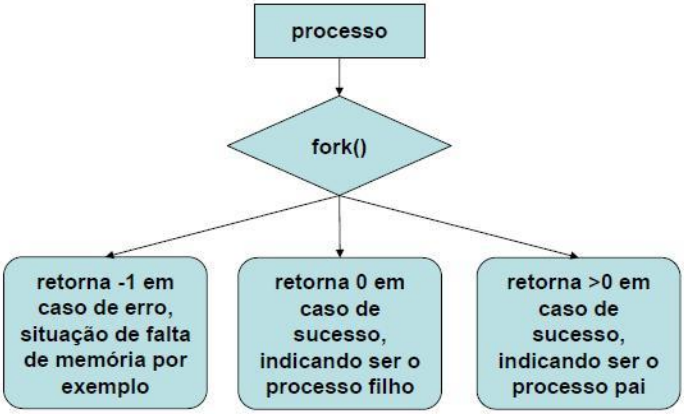

Figura 25 – Chamada de sistema *fork( )*.

- Todos os segmentos de memória compartilhada assinalados;
- Diretório corrente de trabalho;
- Diretório Raiz;
- Máscara de criação de arquivos;

O processo filho possui as seguintes informações diferentes do processo pai:

- PID único dentro do sistema;
- Um PPID diferente. (O PPID do processo filho é o PID do processo pai que inicialmente ativou a função *fork( )*);
- O conjunto de sinais pendentes para o processo é inicializado como estando vazio;
- *Locks* de processo, código e dados não são herdados pelo processo filho;
- Os valores da contabilização do processo obtida pela função *time* são inicializados com zero;
- Todos os sinais de tempo são desligados.

Caso a função *fork( )* retorne 0 (zero), está se executando o processo filho. Caso a função retorne um valor diferente de 0 (zero), mas positivo, o processo pai está sendo executado. O valor retornado representa o PID do processo filho criado. A função retorna -1 em caso de erro (provavelmente devido a se ter atingido o limite máximo de processos por usuário configurado no sistema) [53].

Os Sistemas Operacionais modernos implementam uma semântica de *copy-on-write*, uma modelo utilizado para aumentar a performance evitando escrever todos os dados do aplicativo no momento do *fork( )*, deixando essa tarefa apenas no momento em que as informações da memória são alteradas, ou seja, quando uma cópia do programa é criada. Na verdade, o sistema operacional sinaliza a parte da memória em comum e conforme há alteração realiza cópias. A aplicação desse processo é transparente, é como se a cópia da memória tivesse sido realizada desde o momento da chamada do *fork( )*, mas na verdade fica a cargo do sistema operacional solicitar essa cópia somente quando dados são escritos. Desta maneira grandes conjuntos de dados de aplicativos não precisam ser duplicados caso sejam utilizados apenas para leitura e não sejam alterados por nenhum programa. Muitos benefícios são obtidos com este modelo, como performance e economia de memória [54].

No capítulo que segue, apresenta-se com mais detalhes as estratégias de programação paralela adotadas nas implementações dos métodos *N-Scheme-SOR* e *N-Scheme-GC*.

4 PROPOSTAS DE PARALELIZAÇÃO DO MÉTODO *N-Scheme*

Neste capítulo serão apresentadas as técnicas adotadas para divisão da malha de EF de forma que problemas eletromagnéticos possam ser resolvidos através de técnicas de paralelização. Assim, apresenta-se também, as estratégias de implementação do método *N-Scheme* por troca de mensagens, utilizando a biblioteca MPI para execução em *clusters*, e as estratégias de implementação em computadores *multicore* utilizando a função *fork( )*.

4.2 A IMPLEMENTAÇÃO DO MÉTODO *N-SCHEME* EM PARALELO

Métodos numéricos podem ser empregados juntamente com o contexto de arquiteturas paralelas e programação paralela para obtenção de resultados com ganho de performace. Para que isso seja possível, antes da modelagem computacional do problema a ser resolvido, é necessário a identificação dos fatores que influenciam de maneira significante no problema, obtido através de sua modelagem matemática. A essência dos MEF está na discretização do contínuo, o que torna um problema "finito" e capaz de ser solucionado computacionalmente.

A proposta de implementar o método *N-Scheme* utilizando um paradigma de programação paralela leva à análise de uma série de possibilidades para a divisão do problema em subproblemas que possam ser resolvidos de forma paralela. Isto é feito para acelerar a obtenção dos resultados e/ou diminuir o número de iterações necessárias para convergência do mesmo.

O método *N-Scheme* consiste em resolver o sistema matricial $Ax=b$ através de operações algébricas que são idênticas ao método iterativo de Gauss-Seidel. Esse, por sua vez, apresenta um nível de granulação muito fino, ou seja, a existência de dependência entre as iterações requer um grande número de comunicação entre os processos. Além disso, o método apresenta dificuldade na distribuição equilibrada de tarefas entre os processadores.

Logo, a proposta é dividir a malha de EF de acordo com o número de processos desejado. A divisão da malha deve ser feita tal que se possa distribuir partes da malha entre vários processos sendo que cada um deles utiliza um diferente processador paralelo para calcular, de

forma simultânea, o potencial nos nós incógnitas utilizando o método *N-Scheme*.

4.3 A COMUNICAÇÃO ENTRE OS PROCESSOS

As mensagens enviadas por um processo a outro geralmente são entregues ao destinatário conforme a ordem de envio. No entanto, esta ordem pode ser alterada caso o processo receptor dê prioridade por receber mensagens de um determinado tipo, por exemplo, prioridade em receber mensagens do tipo MPI_CHAR. Mas, para as técnicas adotadas neste trabalho, os processos, emissor e receptor, não terão prioridades por tipos de mensagens.

Um contexto da comunicação foi implementado para se possibilitar a criação de um ambiente distinto e separado para passagem de dados entre os processos, sendo que cada aplicação terá um único contexto [32]. Um exemplo típico que sugere o uso de contexto é a necessidade de se garantir que uma mensagem passada em um determinado momento da aplicação não seja incorretamente interceptada em outro momento da aplicação. A principal questão aqui é garantir que uma determinada mensagem possa trafegar corretamente pelo cluster sem haver perda em seu conteúdo. Uma aplicação MPI executada em um cluster possuirá o seu próprio contexto de comunicação, um ambiente fechado para realizar a troca de dados entre os processos desta aplicação, não interferindo em nenhum momento com uma outra aplicação MPI que possa estar sendo executada no mesmo *cluster* [33].

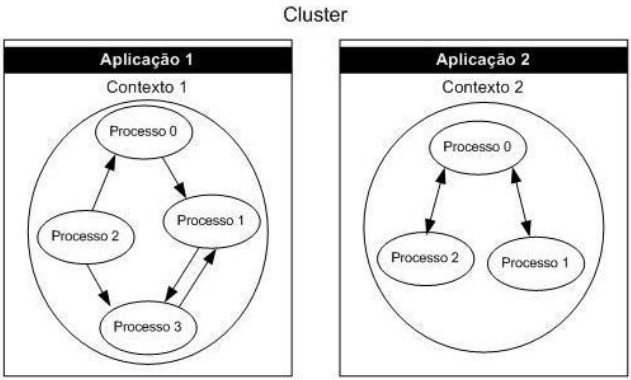

Figura 26 – Contexto de uma aplicação MPI.

A Figura 26 mostra como é a separação feita pela biblioteca MPI para diferenciar duas aplicações MPI rodando em um mesmo *cluster*.

4.3.1 Comunicação Ponto-a-Ponto

Para a comunicação ponto-a-ponto utilizou-se as funções MPI_SEND para o envio das mensagens pelo processo mestre aos demais processos e estes por sua vez, utilizam a função MPI_RECV para recebimento das mensagens. Foi utilizada a comunicação bloqueante (blocking); o processo que receberá a mensagem fica "esperando" até a mensagem chegar, como mostra a Figura 27(a). Ou seja, a biblioteca MPI irá realizar esta passagem de mensagem, mas só irá liberar o processo 1 quando o processo 2 tiver terminado de receber a mensagem. Na Figura 27(b), a biblioteca libera o processo 1 instantaneamente, sem que o processo 2 tenha recebido ainda a mensagem [33].

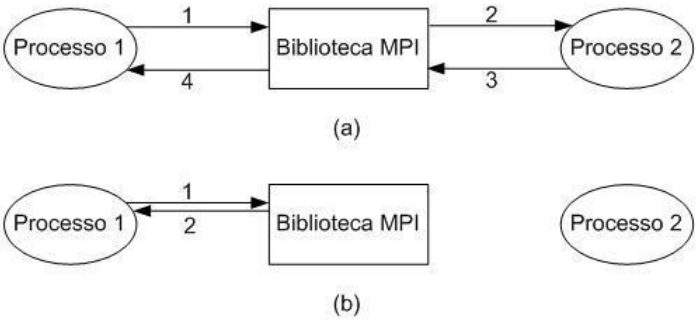

Figura 27 – (a) Passagem de Mensagem Bloqueante e (b) Passagem de Mensagem Não Bloqueante [33].

Ao final, um único processo, o mestre, será responsável por coletar os resultados dos *p* processos, e, para isso, utiliza-se uma função de comunicação coletiva.

4.3.2 Comunicação Coletiva

Para a comunicação coletiva utilizou-se as funções MPI_BCAST (comunicação bloqueante) para o envio de uma mesma mensagem pelo processo mestre a todos os demais processos.

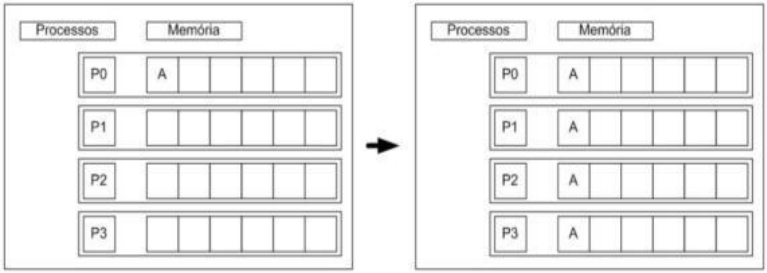

Figura 28 – Exemplo do funcionamento da instrução MPI_Bcast [33].

Utilizou-se a função MPI_GATHER (comunicação bloqueante) para reunir as mensagens vindas do grupo de processos e concatená-las no processo destino.

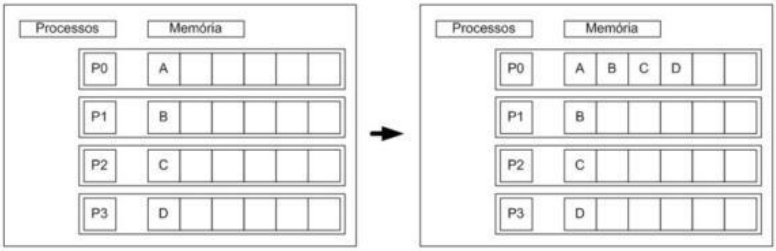

Figura 29 – Exemplo do funcionamento da instrução MPI_Gather [33].

4.4 ESTRATÉGIAS DE PARALELIZAÇÃO DO MÉTODO *N-Scheme* COM GAUSS-SEIDEL E COM GRADIENTE CONJUGADO

4.4.1 Primeira proposta de paralelização: *N-Scheme-SOR*

Para a implementação do método são necessárias algumas informações sobre a malha de EF geradas pelo malhador. Geralmente, o

malhador armazena a topologia da malha através de um vetor, como mostra a Tabela 1. Os dados nela apresentados são referentes à malha apresentada na Figura 3, onde se tem uma malha 2D de elementos retangulares. A técnica é utilizada particularmente em problemas onde o sistema a ser resolvido é grande, como para problemas em 3D. Para efeitos didáticos, demostrar-se-á alguns detalhes de implementação em 2D.

Tabela 1 – Nós por elemento **(*ktri*)**

Elementos	**Nós**
a	1, 5, 6, 2
b	2, 6, 7, 3
c	3, 7, 8, 4
…	…
i	11, 15, 16, 12

Tabela 2 – Elementos por nó **(*nne*)**

Nós	**Número de Elementos (*nn*)**	**Elementos (*en*)**
1	1	a
2	2	a, b
3	2	b, c
…	…	…
6	4	a, b, d, e
…	…	…
16	1	i

4.4.1.1 A divisão do problema

Problemas eletromagnéticos em 2D são utilizados como testes na validação dos resultados. Entretanto, tem-se interesse em resolver problemas eletromagnéticos em 3D, onde a solução do sistema $Ax=b$ se torna complexa à medida que aumentamos o tamanho e a complexidade do problema. A complexidade está diretamente relacionada com a utilização da memória do computador, tempo de processamento e convergência. Apresentar-se-á aqui uma técnica de divisão do problema em subproblemas que são resolvidos de forma simultânea utilizando os conceitos da programação paralela.

4.4.1.2 A geometria

A geometria utilizada para os experimentos é um cubo composto por um único material e potenciais impostos nos nós da superfície superior (v_s =1000) e inferior (v_i = -1000). Trabalhou-se com uma malha de EF de elementos hexaédricos gerados no próprio programa. A escolha por esse tipo de malha se fez necessária devido a problemas encontrados na geração de malhas geradas pelo I-DEAS, contendo milhões de elementos. Assim, desenvolveu-se um programa para resolver um problema eletrostático, calculando a diferença de potenciais nos nós incógnitas da malha.

Para o estudo de caso em questão, a técnica de divisão do problema é feita de acordo com a geometria. Neste caso, adotou-se um número par de divisões que é feita da seguinte forma:

- De acordo com as coordenadas do cubo, faz-se cortes no plano (*x, z*) e cortes paralelos ao plano (*x, y*). Esses cortes dividem a geometria em um número par de regiões de tamanhos iguais;
- Como a geometria é dividida em números pares de regiões, adotou-se para a geração da malha um número de nós divisível por dois, quatro e seis para distribuir o conjunto de nós de forma equilibrada entre os processos;
- Em seguida, cria-se um vetor para cada uma das regiões contendo os seus respectivos nós (*malha1, malha2, ..., malhan.*);
- Esses vetores, contendo os nós de cada região, serão enviados aos processos assim como o vetor de potenciais imposto no problema;
- A matriz *nne* (Tabela 2) - elementos por nó incógnita *n* - também é dividida de acordo com o vetor de nós de cada região. Essas matrizes também serão enviadas aos processos de acordo com os nós das respectivas malhas (*nnem1, nnem2, ..., nnemn.*).

O método *N-Scheme* trabalha com a contribuição dos elementos no nó incógnita. Como optou-se por uma malha de EF regular, a matriz de contribuição ou de rigidez para esse estudo de caso é sempre a mesma, observando que não há materiais diferentes na geometria do problema.

4.4.1.3 Aplicação da programação paralela com MPI

Adotou-se o modelo mestre-escravo, onde o mestre coordena as tarefas, recebendo sua parte de dados e realizando sua parte de cálculo. Utilizou-se da comunicação MPI ponto-a-ponto no modo *standard* com chamadas às rotinas de forma bloqueante.

O processo mestre envia aos *p-1* processos escravos o vetor de nós (*malha1, malha2, ..., malhan*) de cada região e as matrizes *nne* (*nnem1, nnem2, ...,nnemn*). O mestre também é responsável pelo envio do vetor de potenciais *v* aos processos escravos. Cada processo escravo recebe a sua malha e todo o vetor de potenciais *v* e utiliza um diferente processador paralelo para calcular os potenciais sobre a parcela de dados (nós) que recebeu, como mostra o esquema da Figura 30.

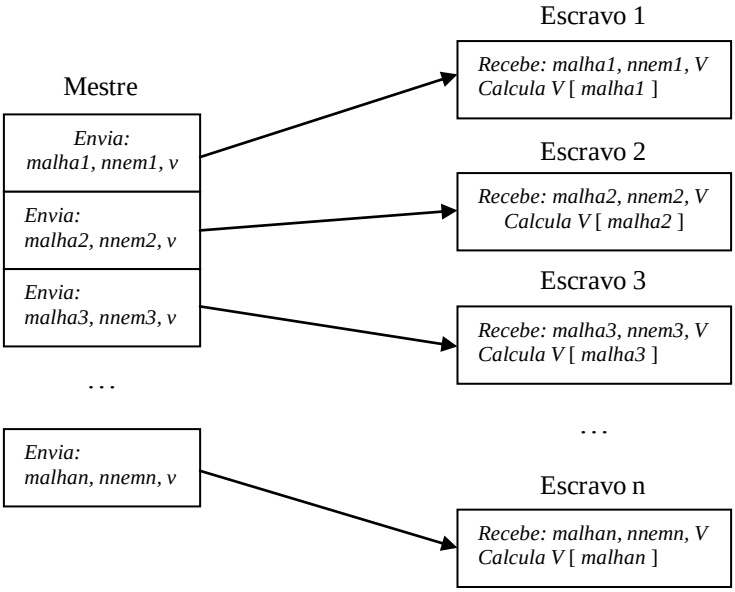

Figura 30 – Divisão das tarefas entre *n* processos.

Cada processo é responsável por calcular o valor do potencial no vetor *v* equivalente aos nós recebidos, os demais valores não são utilizados. Isso significa que os processos não calculam o potencial no mesmo nó, ou seja, os nós recebidos pelos processos são distintos. O processamento dos escravos ocorre simultaneamente, e cada um executa o método *N-Scheme* sobre os potenciais dos nós que recebeu. Após

finalizado o cálculo por cada um dos processos, o mestre reúne os resultados dos potenciais (o vetor *v*) que cada processo escravo calculou e testa a convergência. Se não atingiu a convergência, o mestre reenvia os dados (vetor de nós e vetor *v*) aos processos escravos para novo cálculo, sendo o vetor *v* já atualizado. Quando atingir a convergência, o mestre envia um comando de finalização aos escravos e exibe os resultados.

Todos os processos recebem o vetor de potencias *v* e preenchem as posições desse vetor que correspondem aos nós da sua malha. Supondo-se que o processo 1 calcula o potencial no nó 581, ele colocará na posição 581 do vetor *v* este potencial. Nenhum outro processo calculará o potencial nesse nó. Os processos só terão acesso aos novos valores de *v*, calculados também pelos demais processos, na próxima iteração, quando o vetor *v* é atualizado pelo processo mestre com todos os potencias incógnitas calculados.

Esta técnica foi aplicada em problemas em 3D apresentando resultados satisfatórios quanto à solução numérica obtida assim como os tempos de processamento. Os resultados das execuções quanto à solução encontrada em paralelo comparada à solução através do método sequencial, assim como os tempos de execução dos programas serão apresentados e discutidos no próximo capítulo.

4.4.2 Segunda proposta de paralelização: *N-Scheme-GC*

A diferença significativa para as duas estratégias de implementação está no fato de que, o método modificado possui apenas um *loop* para os elementos, diferente do método original que precisava de dois *loops*, um para os nós e outro para os elementos que rodeiam esse nó, a "célula", como descrito do capítulo 2. Assim, uma nova estratégia de paralelização foi necessária, pois a forma de divisão do problema anteriormente adotado não pôde ser utilizada.

4.4.2.1 Aplicação da programação paralela com MPI

Analisando o algoritmo modificado para uso com GC, percebeu-se que não é necessária uma estratégia de divisão do problema usando a geometria, os nós ou os elementos.

Na construção do algoritmo temos a matriz *ktri* (Tabela 1) que contém os nós de cada elemento da malha de EF e isso é suficiente para

construir o esquema de paralelização para o *N-Scheme* modificado. Isto porque o método *N-Scheme-GC* utiliza apenas um *loop* para os elementos. Observando o algoritmo (Figura 2), o *N-Scheme* é aplicado em dois momentos sendo que, o de maior importância computacional está dentro do *loop* das iterações, e é aqui que se tem interesse em aplicar a paralelização. Assim, como a estrutura depende de um *loop* para os elementos, e as informações utilizadas no processo de cálculo está na matriz *ktri*, a ideia foi simplesmente dividir a matriz *ktri* entre os processos. Para a geração da malha de EF, seguiu-se o princípio de ter um número de elementos que seja divisível pelo número de processadores utilizados, onde cada processador receba partes da matriz *ktri* de forma equilibrada.

Portanto, seguindo o algoritmo da Figura 2, cada processo escravo recebe uma parcela da matriz *ktri*, calcula o valor de *q* referente aos elementos usando o *N-Scheme* e retorna os resultados para o processo mestre. Como alguns elementos possuem alguns nós em comum, no momento da junção dos resultados de *q* recebidos de cada escravo, é realizada a soma destes valores pelo mestre. Feito isto, o mestre calcula um novo vetor *q*, calcula os valores dos potenciais *v*, calcula o erro e o processo continua até atingir a convergência.

A utilização de MPI em *cluster* de computadores para a método com GC, aplicada a problemas em 3D, não apresentou resultados satisfatórios quanto ao tempo de processamento. Os tempos em sequencial e em paralelo ficam muito próximos, sendo que o tempo em paralelo não é melhor que o tempo em sequencial. Isso se deve a granularidade do problema (fina) e a comunicação: quanto de transferência de dados (banda e latência) é necessário? Já os valores numéricos dos potenciais correspondem com o esperado.

Os resultados das execuções quanto à solução encontrada em paralelo comparada à solução através do método sequencial, assim como os tempos de execução dos programas serão apresentados e discutidos no capítulo 5.

4.4.2.2 Aplicação da programação paralela *Multicore*

Como a estratégia de utilizar MPI em *cluster* de computadores para o método *N-Scheme-GC* não apresentou resultados satisfatórios quanto ao tempo de processamento, partiu-se para o estudo de uma nova estratégia que envolvesse o paradigma de programação paralela, mas,

com outro foco: em ambientes multiprocessados com memória compartilhada.

Os programas, inicialmente, foram desenvolvidos e simulados utilizando a linguagem de programação FORTRAN 77. Com o impasse de não conseguir os resultados de tempo de processamento em paralelo melhores que no sequencial, optou-se pela implementação do algoritmo em C, o qual não trouxe mudanças significativas.

Iniciaram-se assim, os estudos sobre as formas de programação paralela em ambientes *multicore*. Aqui, partes da matriz *ktri* que contém informações dos nós que contribuem em cada elemento da malha (Tabela 1) são acessadas pelos processos criados nas chamadas da função *fork ( )*, conforme Figura 31.

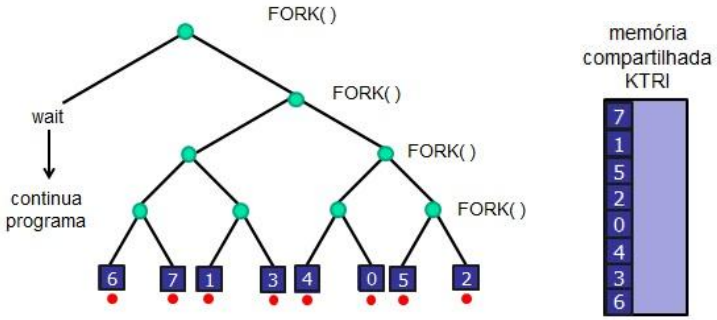

Figura 31 – Criação de processos com a função *fork ( )*.

A primeira chamada a função *fork ( )* cria dois processos sendo que um ficará aguardando a finalização de todos os demais para dar continuidade ao programa. Cada chamada de *fork ( )* cria 2 processos filhos que são cópias do processo pai. Na Figura 31, temos a criação de 8 processos que acessam diferentes posições da matriz *ktri*. Cada processo aplica o método *N-Scheme* para calcular os potenciais nos nós incógnitos.

Nestes ambientes (sistemas *multicore*) é preciso enfrentar alguns desafios nas execuções dos programas, como:

a) o sistema operacional permite que mais de uma atividade ocorra ao mesmo tempo;

b) gerenciar atividades diferentes pode gerar problemas de sincronização, comunicação, balanceamento de carga e escalabilidade.

Assim, como medir esses problemas? A resposta para essa pergunta veio através da análise dos resultados obtidos em várias execuções dos programas, o que levou a melhorias na implementação do código paralelo levando a solucionar os desafios citados acima.

Nesses experimentos atentou-se para:

- Sincronização: como coordenar a atividade de dois ou mais processos;
- Comunicação: quanto de transferência de dados (banda e latência) é necessário;
- Balanceamento de carga: como distribuir o trabalho entre os processos de forma a não deixar nenhum ocioso;
- Escalabilidade: como fazer bom uso dos processos quando o sistema computacional aumentar.

Todas essas questões foram solucionadas utilizando comandos específicos que atendessem a todos os requisitos necessários para um bom desempenho do programa em paralelo. Esta estratégia foi aplicada em problemas em 3D apresentando resultados satisfatórios quanto à solução numérica obtida assim como os tempos de processamento sequencial comparado com o paralelo. Os resultados das execuções dos programas quanto à solução encontrada em paralelo comparada à solução através do código sequencial, assim como os tempos de execução dos programas serão apresentados e discutidos no próximo capítulo.

4.5 CONSIDERAÇÕES SOBRE AS PROPOSTAS DE PARALELIZAÇÃO

A primeira proposta de implementação do método *N-Scheme-SOR* é baseado nos nós da malha de EF e a divisão do problema aplicada e implementada levou isso em consideração. A implementação desta proposta utilizando a troca de mensagens (MPI) em *clusters* apresentou resultados satisfatórios para malhas mais refinadas. Com isso foi possível obter as medidas de desempenho dos programas em paralelo comparados com o sequencial.

A segunda proposta de implementação, para o método *N-Scheme-GC*, foi muito mais simples que a primeira. Esta utiliza apenas dois *loops*, um para as iterações e outro para o método *N-Scheme*. Enquanto

que na primeira proposta se tem três *loops*, um para as iterações e dois para o método *N-Scheme*. Isto tornou a implementação da segunda proposta mais simples e consequentemente mais rápida com a diminuição de um *loop*. Além disso, a adaptação do método *N-Scheme-GC* em sua forma sequencial atinge um tempo computacional muito menor, comparado ao *N-Scheme-SOR*. Desta forma, a implementação do método utilizando MPI não resultou em resultados satisfatórios quanto ao tempo de processamento sequencial comparado com o paralelo. Considerou-se então a implementação paralela em máquinas multiprocessadas.

5 AMBIENTES E RESULTADOS EXPERIMENTAIS

Este capítulo apresenta os resultados obtidos com a paralelização do método *N-Scheme-SOR* e *N-Scheme-GC*, proposto neste trabalho de tese. Serão apresentados os resultados de tempo de processamento obtidos de forma sequencial e em paralelo, principal objeto de estudo deste trabalho.

5.1 ESTRATÉGIAS ADOTADAS PARA AS EXECUÇÕES EM *CLUSTERS* E EM ARQUITETURA MULTICORE

5.1.1 Resultados dos Métodos *N-Scheme-SOR* e *N-Scheme-GC* usando MPI

O programa foi desenvolvido utilizando a linguagem de programação FORTRAN 77 em conjunto com a biblioteca de comunicação MPI. Os resultados das execuções foram obtidos trabalhando nos *clusters* da *Sharcnet*, uma instituição acadêmica Canadense, que disponibilizou o acesso a sua rede de computadores de alto desempenho.

A *Sharcnet* disponibilizou a utilização de até oito processadores para a realização das execuções dos programas. A técnica de programação paralela adotada foi a de mestre-escravo, onde um dos processadores faz o papel de mestre e os demais são os escravos. O mestre fica responsável pelo gerenciamento e distribuição das tarefas, enquanto que os escravos são responsáveis pelo cálculo dos potenciais nos nós da malha, utilizando o *N-Scheme*. Nesta proposta, o mestre não trabalha no cálculo dos potenciais, apenas distribui as tarefas, reúne os resultados, calcula o erro e verifica a convergência.

Na *Sharcnet* há vários *clusters* que suportam a compilação e execução de diversas aplicações. A máquina escolhida para os experimentos foi aquela que suporta a compilação e execução de programas escritos em FORTRAN 77 e com suporte a programação paralela por troca de mensagens, MPI. Portanto, todas as execuções foram aplicadas em problemas 3D e os resultados apresentados são para um número de processadores variável.

O principal objetivo deste trabalho de tese é mostrar que, com os experimentos apresentados, resultados de tempo de processamento e medidas de desempenho são alcançados. Os resultados de tempo de

processamento obtidos na execução do programa escrito de forma sequencial foram conseguidos utilizando o comando *DTIME*. Já para o tempo de processamento na execução do programa em paralelo foi utilizado o comando *MPI_WTIME*, retornando o tempo de processamento em segundos. Para o programa sequencial, como as malhas de EF utilizadas atingem a ordem de milhões de elementos, optou-se apenas pela execução no *cluster* com sistema operacional HP Linux XC 3.1.

As malhas de EF foram geradas no próprio código devido a problemas na criação das mesmas em outros softwares como citado em capítulo anterior devido as suas proporções. As malhas de EF foram geradas na ordem de milhões de elementos para ter-se resultados significativos quanto ao tempo de processamento em paralelo comparado com o tempo em sequencial.

Para os testes das duas propostas de paralelização, *N-Scheme-SOR* e *N-Scheme-GC*, adotou-se como critério de parada uma precisão de 10^{-5} e para a primeira proposta, um fator de relaxação $R_{final} = 0{,}96$. A comunicação entre os processos é feita a cada iteração, ou seja, a cada iteração o mestre envia os dados e recebe os resultados, até que atinja a convergência desejada.

As tabelas apresentadas neste capítulo mostram os resultados do tempo de processamento em paralelo e em sequencial executados no *cluster Requin* e *Kraken* [41] respectivamente. Os *clusters* possuem as seguintes configurações de hardware:

Requin:

- sistema operacional HP Linux XC 3.1
- rede de interconexão *quadrics elan4*
- memória de 4 ou 8 GB para os nós
- processador de 2.6 GHz
- 2 cores, 1 sockets x 2 cores per socket

Kraken:

- sistema operacional CentOS 5.4
- rede de interconexão *myrinet*
- memória de 4, 8 ou 32 GB para os nós
- processador de 2.2 GHz
- cores, 2 sockets x 2 cores per socket

A escolha do *cluster Kraken* para executar o programa sequencial foi recomendação da *Sharcnet*.

Para todas as tabelas que serão apresentadas nos próximos tópicos, as malhas serão referenciadas pela variável *Nx* (igual a *Ny* e *Nz*)

- Nº de elementos = *Nx* x *Ny* x *Nz*
- Nº de nós = (*Nx+1*)x(*Ny+1*)x(*Nz+1*)

5.1.1.1 *N-Scheme-SOR* usando MPI

Os resultados apresentados na Tabela 3, o tempo sequencial foi medido executando-se o programa no *cluster Kraken*.

Tabela 3 – Tempo de execução **sequencial** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
125	182	275	1.953.125	2.000.376
131	206	302	2.248.091	2.299.968
155	625	408	3.723.875	3.796.416
173	810	489	5.177.717	5.268.024
191	1.925	572	6.967.871	7.077.888
203	1.834	628	8.365.427	8.489.664
221	2.799	715	10.793.861	10.941.048
227	3.127	774	11.697.083	11.852.352

A Tabela 3 e o gráfico da Figura 32 mostram os resultados do tempo de execução do código sequencial, o número de iterações de acordo com o tamanho da malha.

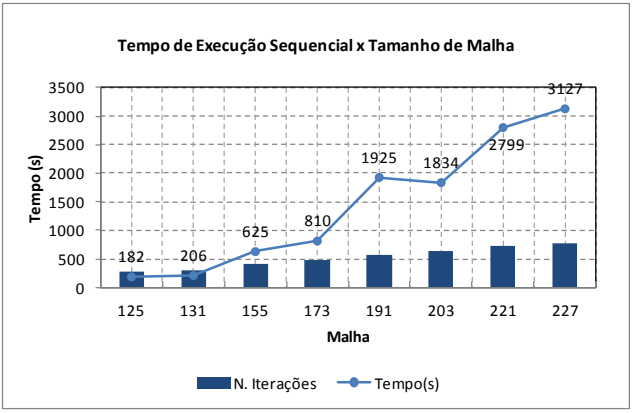

Figura 32 – Tempo de execução sequencial versus tamanho de malha e número de iterações.

Tabela 4 – Tempo de execução **paralelo (2p)** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
125	190	361	1.953.125	2.000.376
131	168	305	2.248.091	2.299.968
155	347	369	3.723.875	3.796.416
173	576	414	5.177.717	5.268.024
191	995	448	6.967.871	7.077.888
203	954	461	8.365.427	8.489.664
221	1.422	475	10.793.861	10.941.048
227	1.418	478	11.697.083	11.852.352

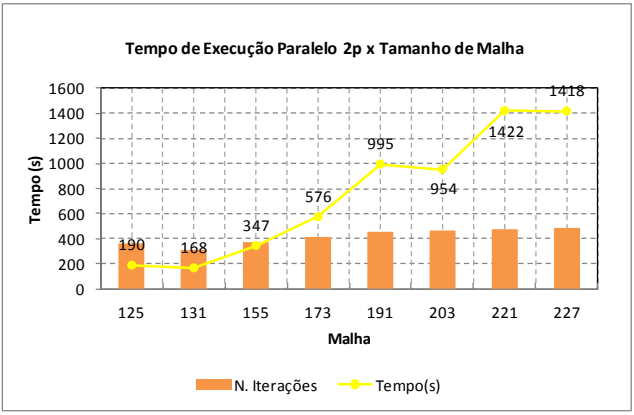

Figura 33 – Tempo de execução paralelo (2 processos) versus tamanho de malha e número de iterações.

A Tabela 4 e o gráfico da Figura 33 mostram os resultados do tempo de execução do código paralelo utilizando 2 processos e o número de iterações de acordo com o tamanho da malha.

Tabela 5 – Tempo de execução **paralelo (4p)** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
125	118	290	1.953.125	2.000.376
131	157	306	2.248.091	2.299.968
155	318	369	3.723.875	3.796.416
173	450	414	5.177.717	5.268.024
191	700	449	6.967.871	7.077.888
203	753	460	8.365.427	8.489.664
221	1.077	474	10.793.861	10.941.048
227	1.165	477	11.697.083	11.852.352

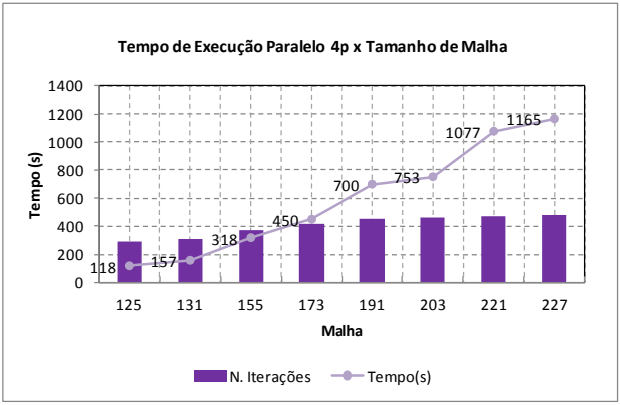

Figura 34 – Tempo de execução paralelo (4 processos) versus tamanho de malha e número de iterações.

A Tabela 5 e o gráfico da Figura 34 mostram os resultados do tempo de execução do código paralelo utilizando 4 processos e o número de iterações de acordo com o tamanho da malha.

Tabela 6 – Tempo de execução **paralelo (6p)** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
125	113	291	1.953.125	2.000.376
131	141	306	2.248.091	2.299.968
155	311	369	3.723.875	3.796.416
173	445	415	5.177.717	5.268.024
191	699	449	6.967.871	7.077.888
203	826	460	8.365.427	8.489.664
221	1.136	473	10.793.861	10.941.048
227	1.080	476	11.697.083	11.852.352

Já a Tabela 6 e o gráfico da Figura 35 mostram os resultados do tempo de execução do código paralelo utilizando 6 processos e o número de iterações de acordo com o tamanho da malha.

Observando os resultados apresentados nas tabelas e gráficos, para uma malha na ordem de 2 milhões de nós, nota-se que o tempo de processamento em paralelo comparado com o sequencial, já possui considerável diferença. Nota-se também que, para a malha 125 o tempo de execução com 2 processos é maior que o tempo em sequencial. O tempo melhora ao se utilizar 4 e 6 processos bem como o número de iterações que se torna menor comparado ao sequencial.

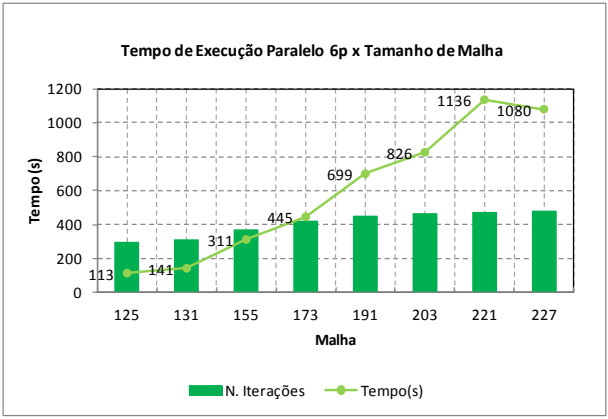

Figura 35 – Tempo de execução paralelo (6 processos) versus tamanho de malha e número de iterações.

Para os demais experimentos, onde a ordem da malha é maior, o tempo de processamento em paralelo vai se tornando melhor comparado ao tempo em sequencial. Nota-se também que, em todas as execuções, a utilização de quatro ou seis processadores apresenta um tempo de processamento melhor que ao se utilizar apenas dois processadores. Pode-se analisar também, na Figura 36, que o tempo de processamento em paralelo utilizando 4 ou 6 processadores apresenta pouca diferença, devido às características e comportamento da rede de interconexão entre as máquinas. Quanto ao número de iterações, observa-se que, à medida que a ordem da malha aumenta as iterações diminuem em paralelo comparado ao sequencial.

Algumas variações no tempo de execução observadas em algumas malhas ocorre devido à patologia e saturação da rede de interconexão. O controle de tráfego de dados na rede é essencial para propiciar condições justas para que todos naveguem com boa velocidade. Apesar de muitos se referirem às atividades citadas como "controle de banda", o nome correto é "controle de tráfego". Banda é a capacidade de transmissão que um meio tem. Essa capacidade é medida em *hertz*. Controle de tráfego significa definir quais pacotes têm prioridade bem como a velocidade poderão trafegar.

Assim, nos experimentos realizados nos *clusters*, o controle de tráfego é gerenciado pela *Sharcnet*.

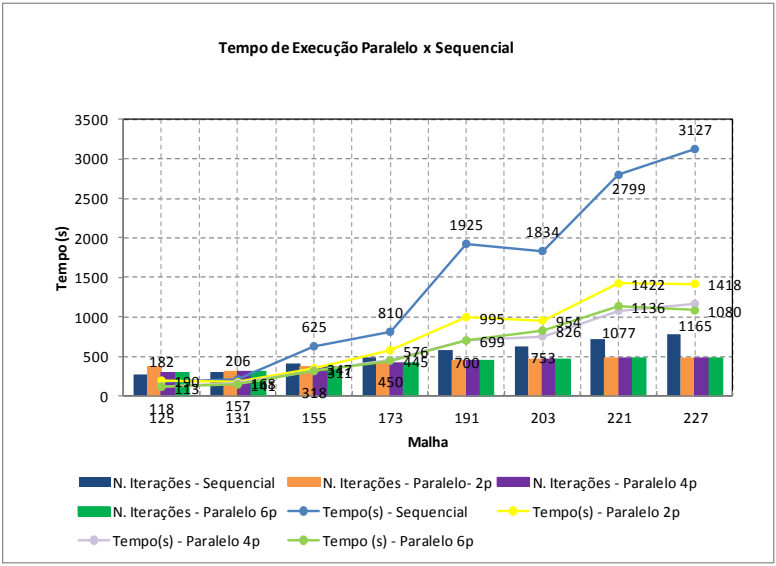

Figura 36 – Tempo de execução para 2, 4 e 6 processadores versus sequencial.

5.1.1.2 Cálculo do *Speedup*

Como visto no capítulo 3, existem múltiplas alternativas para avaliar o desempenho de programas paralelos, mas, geralmente o peso de cada um dos elementos varia de aplicação para aplicação.

Idealmente, o ganho de *speedup* deveria tender a p (número de processadores), que seria o seu valor ideal, como mostra a Tabela 7.

Tabela 7 – Comportamento do Speedup e da Eficiência

Caso	*Speedup*	*Eficiência*
Ideal	$= p$	$= 1$
Real	$< p$	< 1
Exponencial	$> p$	> 1

Para as execuções do método *N-Scheme-SOR*, o comportamento do *speedup* é apresentado no gráfico da Figura 37.

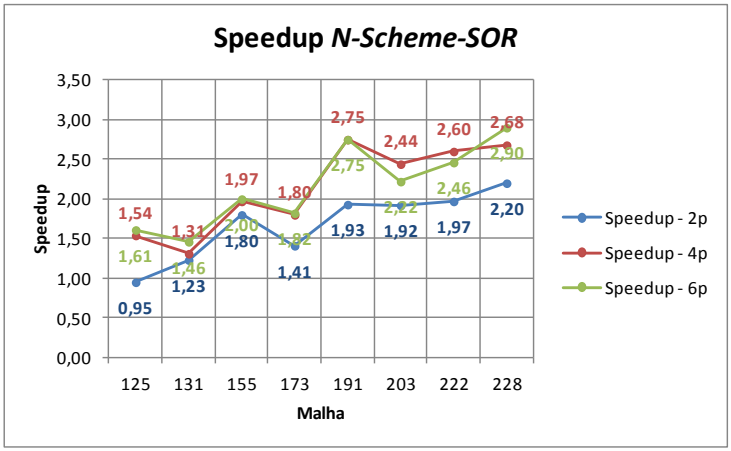

Figura 37 – Comportamento do *speedup* para o *N-Scheme-SOR* usando MPI.

Considerando que um programa paralelo seja executado por **P** processadores e que cada processador é responsável por **1/P** do processamento, o *speedup* máximo será **P**. Entretanto, isto não ocorre na prática devido a fatores como sincronização (coordenação temporal das tarefas) e *overheads* (tempo utilizado para coordenar tarefas, transferir dados, entre outros consumos de tempo) responsáveis por acréscimos no tempo de execução paralelo. Assim, considerando que o método é de granularidade fina (muita comunicação), um *speedup* da ordem de aproximadamente 3 para a maior malha utilizando 6 processos é significativo.

5.1.1.3 *N-Scheme-GC* usando MPI

Como o método *N-Scheme-GC* em sequencial se tornou mais rápido que o método *N-Scheme-SOR*, as execuções já iniciaram com malhas maiores. O tempo sequencial também foi medido executando-se o programa em uma máquina do *cluster*.

Tabela 8 – Tempo de execução **sequencial** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
192	129	95	7.077.888	7.189.057
222	140	110	10.941.048	11.089.567
258	301	128	17.173.512	17.373.979
276	364	137	21.024.576	21.253.933

Tabela 9 – Tempo de execução **paralelo (2p)** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
192	165	95	7.077.888	7.189.057
222	218	110	10.941.048	11.089.567
258	391	128	17.173.512	17.373.979
276	572	137	21.024.576	21.253.933

Tabela 10 – Tempo de execução **paralelo (4p)** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
192	151	95	7.077.888	7.189.057
222	233	110	10.941.048	11.089.567
258	518	128	17.173.512	17.373.979
276	618	137	21.024.576	21.253.933

Tabela 11 – Tempo de execução **paralelo (6p)** x tamanho da malha

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
192	178	95	7.077.888	7.189.057
222	321	110	10.941.048	11.089.567
258	682	128	17.173.512	17.373.979
276	740	137	21.024.576	21.253.933

Como pode ser observado nos resultados apresentados na Tabela 8 à Tabela 11, o tempo de execução do programa em paralelo apresentou, em todos os casos, um tempo muito maior que o apresentado pelo sequencial. Além disso, para todas as malhas testadas, o aumento do número de processadores, não necessariamente apresenta um tempo de processamento melhor. Esse comportamento se deve aos desafios e limitações comentados no capítulo 4, como a granularidade do problema e a comunicação (largura de banda e latência da rede).

5.1.2 Resultados do Método *N-Scheme-GC* em ambiente *Multicore*

Nesta seção serão apresentados os resultados das execuções com o método *N-Scheme-GC* utilizando conceitos de programação paralela *multicore*. Nas execuções do programa, avalia-se o desempenho obtido com sua execução em sequencial e com sua execução em paralelo. O

método não pode ser totalmente paralelizado, porém o cálculo dos nós dos elementos que compõem a malha 3D pode ocorrer paralelamente desde que haja sincronismo no início da execução dos processos. Os experimentos são demonstrados através de tabelas e gráficos (curvas comparativas do tempo de execução e número de iterações). O ambiente utilizado para a execução do programa pode influenciar no desempenho do algoritmo, assim propõem-se alguns cuidados com sua execução e as limitações que possam existir.

As arquiteturas de hardware utilizadas foram todas x86 32bits, com o objetivo avaliar as vantagens e desvantagens oferecidas por ambientes *desktop multicore* como se pode observar na Tabela 12. Grande parte da motivação está no fato de atualmente ser relativamente fácil encontrar processadores *multicore* em computadores Desktop domésticos e corporativos.

Com o objetivo de tentar obter o melhor desempenho possível e o melhor aproveitamento do tempo do processador e dos recursos da máquina, instalou-se a versão Server da Distribuição GNU/Linux Ubuntu 12.04. Essa versão não possui interface gráfica e aplicativos de usuários executando de forma que possam prejudicar as medições. Apenas alguns *daemons* básicos e o próprio *Kernel* é que estão em execução e oferecem pouca influência no desempenho do programa que usaremos para testar o método *N-Scheme-GC*.

Com a vasta variedade de processadores e suas diferentes tecnologias é muito difícil oferecer um comparativo equilibrado entre processadores Intel e AMD como mostra a Figura 38.

As execuções, para o código em paralelo e também para o sequencial, foram realizadas de duas formas: sem a *flag* de otimização na compilação do código paralelo e com a *flag* de otimização.

As *flags* de otimização permitem ao compilador executar diversos procedimentos no código para torná-lo menor ou ainda mais rápido, corrigindo partes do código de maneira a torná-lo mais eficiente. Existem diversos tipos de otimização que o compilador pode realizar e essa opção é passada com a *flag ' –Ox'* aonde *x* é um número como, por exemplo, '*–O3*' (otimização utilizada no *NScheme-GC*).

Para garantir que a execução do código recebesse prioridade do sistema operacional e pudesse ter seu fluxo bem definido, utilizou-se também o comando *nice* do *Linux* para definir o programa com a maior prioridade sobre todos os processos (*nice* –n -20 <comando>) evitando problemas no fluxo e no sincronismo dos processos executados em paralelo [55] [56].

Estas otimizações são próprias do compilador (GCC) que foram usadas nas compilações com as configurações dos computadores 1 e 4, conforme apresentado na Tabela 12.

Tabela 12- Equipamentos disponíveis para os experimentos

Descrição	Computador_1	Computador_2	Computador_3	Computador_4
Processador	Intel Core i7-2600	Intel Core i7 860	AMD Phenom II X4 965	Intel Core i7 3770k
N.° de Cores	4	4	4	4
Clock Máx. p/ Core	3400Mhz	2600Mhz32w1	3400Mhz	4800Mhz (OC)
N.° de Threads	8	8	4	8
Cache	8192 KB	8192 KB	6144 KB	8192 KB
Memória	Corsair XMS 3 TR3X6G1600C9	Kingstone	Corsair XMS 3 TR3X6G1600C9	Samsung (30nm) m379b5273dh0
Barramento Memória	DDR3 1333mhz	DDR3 1333mhz	DDR3 1333mhz	DDR3 2000mhz
Qtd. Memória	4096MB (4GB)	8192MB (8GB)	4096MB (4GB)	8192MB (8GB)
Placa Mãe	Asus P8P67 EVO	Asus Maximus III Gene	Asus M4A7BT-E	Asus Sabertooth Z77
GPU	Nvidia GeForce GTS 250 1024MB	Nvidia GeForce 9800GT 512MB	ATI Radeon HD 5670 776MB	Nvidia GeForce GTX 670 3072MB
Sistema Operacional	GNU/Linux 3.2.0 32bit fornecido pela distribuição Ubuntu Server 12.04 LTS			

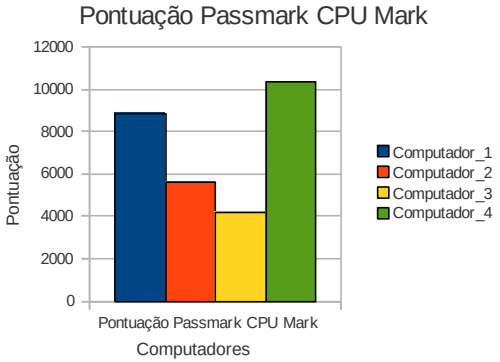

Figura 38 – Comparativo de desempenho dos computadores segundo tabela de *benchmarks* publicada em www.cpubenchmark.net.

Para todas as tabelas que serão apresentadas nos próximos tópicos, as malhas serão referenciadas pela variável *Nx* (igual a *Ny* e *Nz*)

- Nº de elementos = *Nx* x *Ny* x *Nz*
- Nº de nós = (*Nx+1*)x(*Ny+1*)x(*Nz+1*)

5.1.2.1 Execuções sem a *flag* de otimização (computador_1: PC1)

A Tabela 13 e o gráfico da Figura 39 mostram os resultados do tempo de execução do código sequencial, o número de iterações de acordo com o tamanho da malha no computador_1.

Tabela 13 - Tempo de execução **sequencial** x tamanho da malha (não otimizado) PC1

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	28	118	2.097.152	2.146.689
160	50	107	4.096.000	4.173.281
192	127	160	7.077.888	7.189.057
224	279	220	11.239.424	11.390.625
256	635	335	16.777.216	16.974.593
288	810	302	23.887.872	24.137.569
320	1844	498	32.768.000	33.076.161

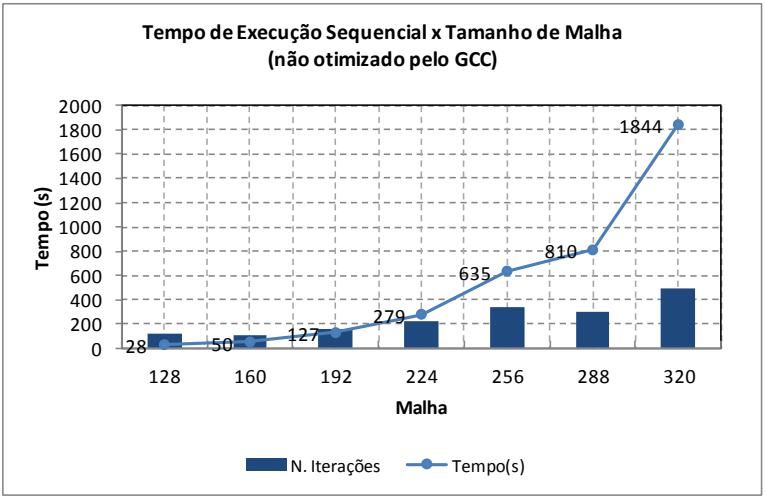

Figura 39 – Tempo de execução sequencial versus tamanho de malha e número de iterações (não otimizado) PC1.

A Tabela 14 e o gráfico da Figura 40 mostram os resultados do tempo de execução do código paralelo com 8 processos, o número de iterações de acordo com o tamanho da malha no computador_1.

Tabela 14 – Tempo de execução **paralelo** x tamanho da malha (não otimizado) PC1

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	12	99	2.097.152	2.146.689
160	29	127	4.096.000	4.173.281
192	82	212	7.077.888	7.189.057
224	128	207	11.239.424	11.390.625
256	300	328	16.777.216	16.974.593
288	249	191	23.887.872	24.137.569
320	1370	775	32.768.000	33.076.161

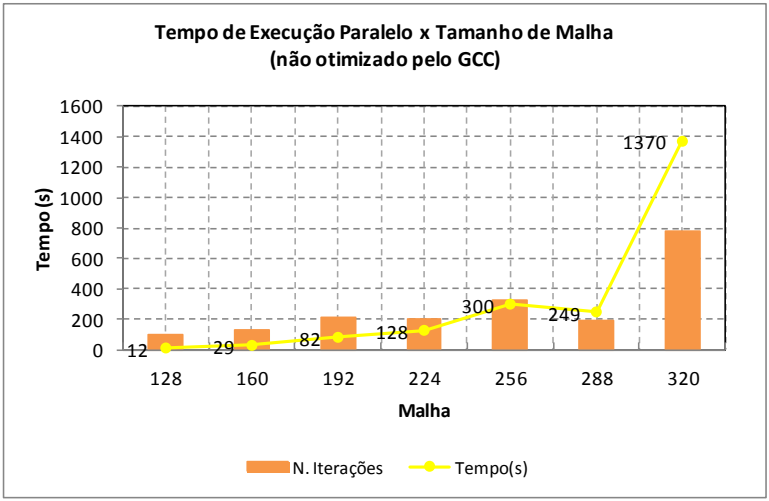

Figura 40 – Tempo de execução paralelo versus tamanho de malha e número de iterações (não otimizado) PC1.

O gráfico da Figura 41 mostra um comparativo do tempo de execução sequencial e paralelo mostradas individualmente na Figura 39 e Figura 40. O comportamento do *speedup* pode ser observado no gráfico da Figura 42.

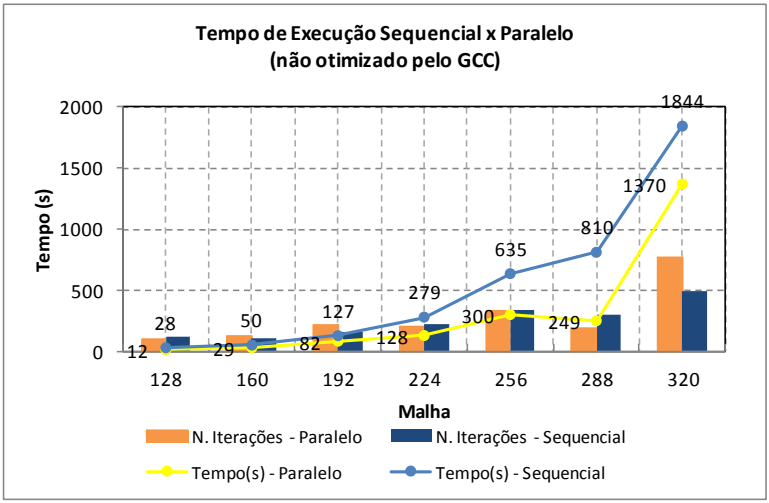

Figura 41 – Tempo de execução paralelo versus sequencial (não otimizado) PC1.

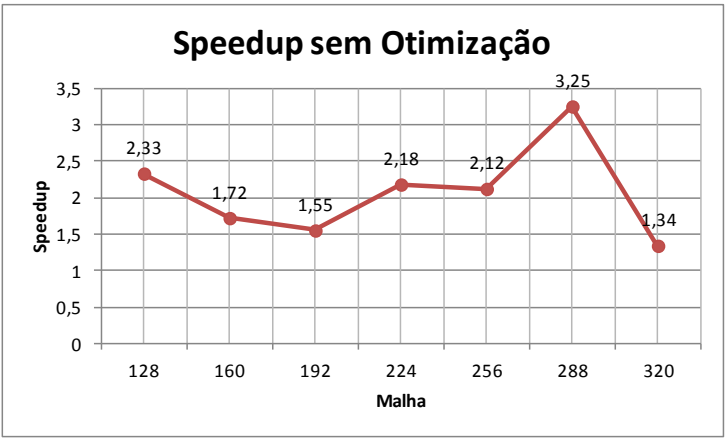

Figura 42– Comportamento do *Speedup* para o método *N-Scheme-GC* (não otimizado) PC1.

5.1.2.2 Execuções com a *flag* de otimização (computador_1: PC1)

A Tabela 15 e o gráfico da Figura 43 mostram os resultados do tempo de execução do código sequencial, o número de iterações de acordo com o tamanho da malha utilizando a *flag* de otimização do compilador no computador_1.

Tabela 15 – Tempo de execução **sequencial** x tamanho da malha (otimizado) PC1

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	7	63	2.097.152	2.146.689
160	17	79	4.096.000	4.173.281
192	36	95	7.077.888	7.189.057
224	65	111	11.239.424	11.390.625
256	116	127	16.777.216	16.974.593
288	190	143	23.887.872	24.137.569
320	270	159	32.768.000	33.076.161

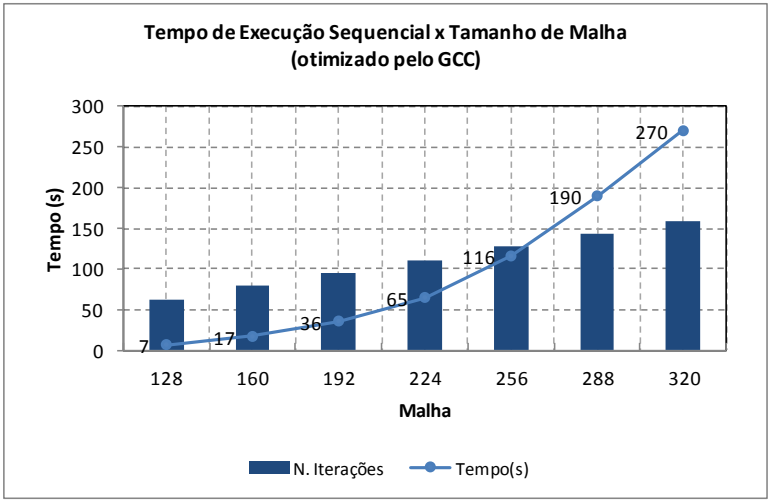

Figura 43 – Tempo de execução sequencial versus tamanho de malha e número de iterações (otimizado) PC1.

A Tabela 16 e o gráfico da Figura 44 mostram os resultados do tempo de execução do código paralelo, o número de iterações de acordo com o tamanho da malha utilizando a *flag* de otimização do compilador no computador_1 com 8 processos.

Tabela 16 – Tempo de execução **paralelo** x tamanho da malha (otimizado) PC1

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	4	63	2.097.152	2.146.689
160	10	79	4.096.000	4.173.281
192	21	95	7.077.888	7.189.057
224	37	111	11.239.424	11.390.625
256	62	127	16.777.216	16.974.593
288	98	143	23.887.872	24.137.569
320	128	159	32.768.000	33.076.161

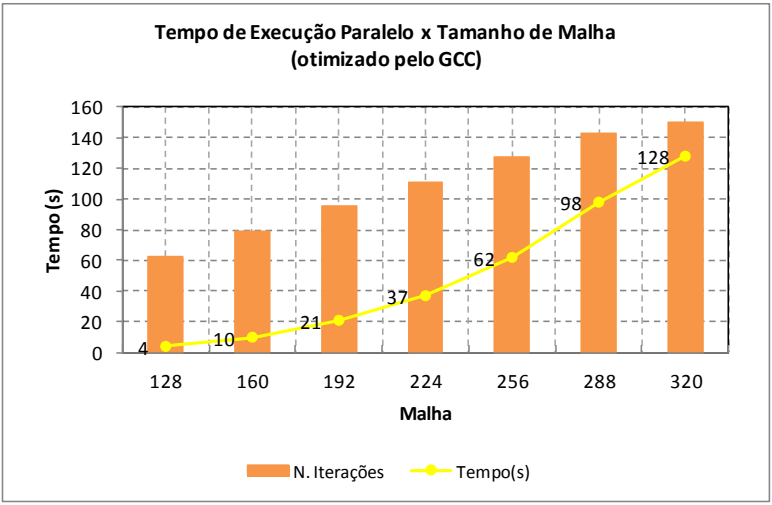

Figura 44 – Tempo de execução paralelo versus tamanho de malha e número de iterações (otimizado) PC1.

O gráfico da Figura 45 mostra um comparativo do tempo de execução sequencial e paralelo mostradas individualmente na Figura 43 e Figura 44. O comportamento do *speedup* pode ser observado no gráfico da Figura 46.

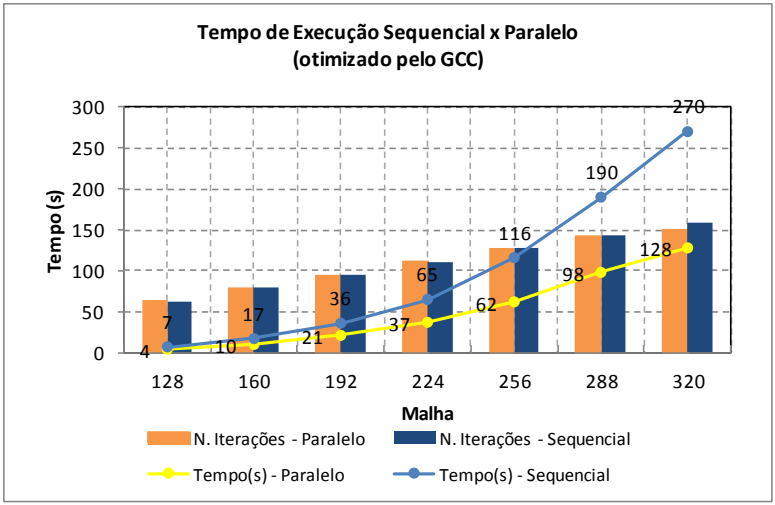

Figura 45 – Tempo de execução paralelo versus sequencial (otimizado) PC1.

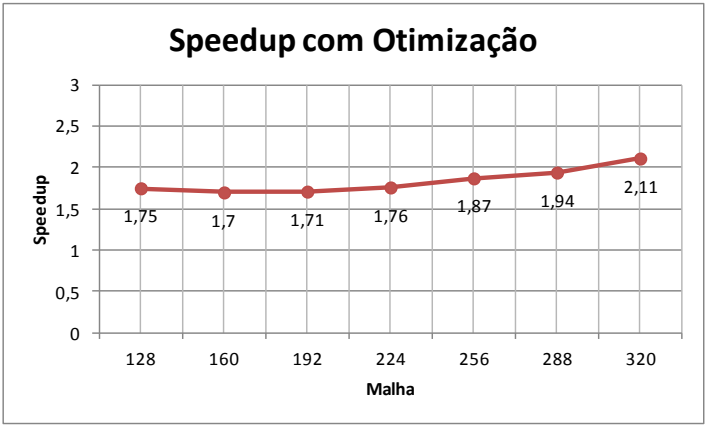

Figura 46 – Comportamento do *Speedup* para o método *N-Scheme-GC* (não otimizado) PC1.

5.1.2.3 Execuções sem a *flag* de otimização (computador_4: PC4)

A Tabela 17 e o gráfico da Figura 47 mostram os resultados do tempo de execução do código sequencial, o número de iterações de acordo com o tamanho da malha no computador_4.

Tabela 17 – Tempo de execução **sequencial** x tamanho da malha (não otimizado) PC4

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	20	118	2.097.152	2.146.689
160	35	107	4.096.000	4.173.281
192	90	160	7.077.888	7.189.057
224	197	220	11.239.424	11.390.625
256	445	335	16.777.216	16.974.593
288	576	302	23.887.872	24.137.569
320	1342	498	32.768.000	33.076.161

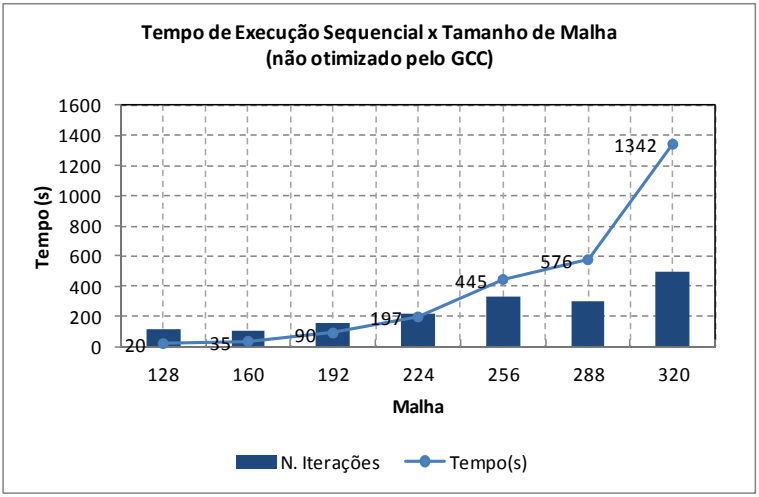

Figura 47 – Tempo de execução sequencial versus tamanho de malha e número de iterações (não otimizado) PC4.

A Tabela 18 e o gráfico da Figura 48 mostram os resultados do tempo de execução do código paralelo com 8 processos, o número de iterações de acordo com o tamanho da malha no computador_4.

Tabela 18 – Tempo de execução **paralelo** x tamanho da malha (não otimizado) PC4

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	9	99	2.097.152	2.146.689
160	20	127	4.096.000	4.173.281
192	59	215	7.077.888	7.189.057
224	90	207	11.239.424	11.390.625
256	198	309	16.777.216	16.974.593
288	172	191	23.887.872	24.137.569
320	658	536	32.768.000	33.076.161

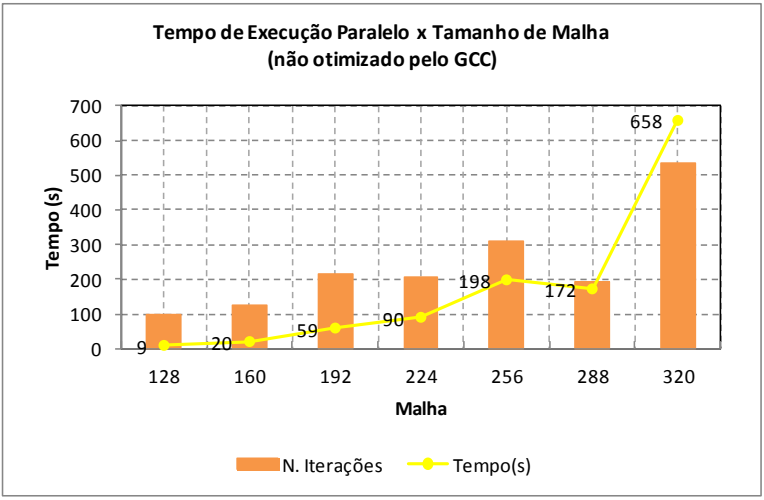

Figura 48 – Tempo de execução paralelo versus tamanho de malha e número de iterações (não otimizado) PC4.

O gráfico da Figura 49 mostra um comparativo do tempo de execução sequencial e paralelo mostradas individualmente na Figura 47 e Figura 48.

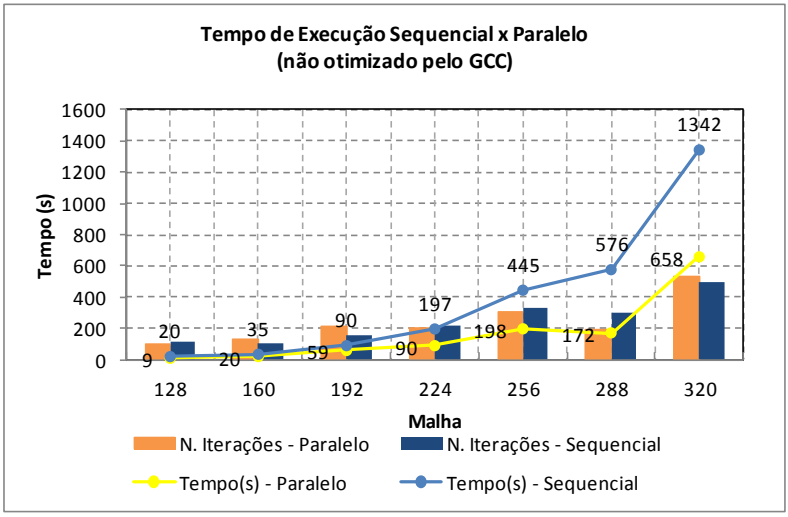

Figura 49 – Tempo de execução paralelo versus sequencial (não otimizado) PC4.

O comportamento do *speedup* pode ser observado no gráfico da Figura 50.

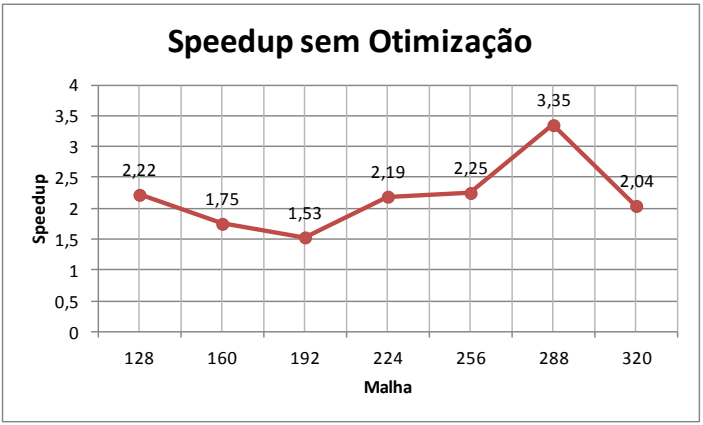

Figura 50 – Comportamento do *Speedup* para o método *N-Scheme-GC* (não otimizado) PC4.

5.1.2.4 Execuções com a *flag* de otimização (computador_4: PC4)

A Tabela 19 e o gráfico da Figura 51 mostram os resultados do tempo de execução do código sequencial, o número de iterações de acordo com o tamanho da malha utilizando a *flag* de otimização do compilador no computador_4.

Tabela 19 – Tempo de execução **sequencial** x tamanho da malha (otimizado) PC4

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	5	63	2.097.152	2.146.689
160	11	79	4.096.000	4.173.281
192	24	95	7.077.888	7.189.057
224	45	111	11.239.424	11.390.625
256	76	127	16.777.216	16.974.593
288	123	143	23.887.872	24.137.569
320	186	159	32.768.000	33.076.161

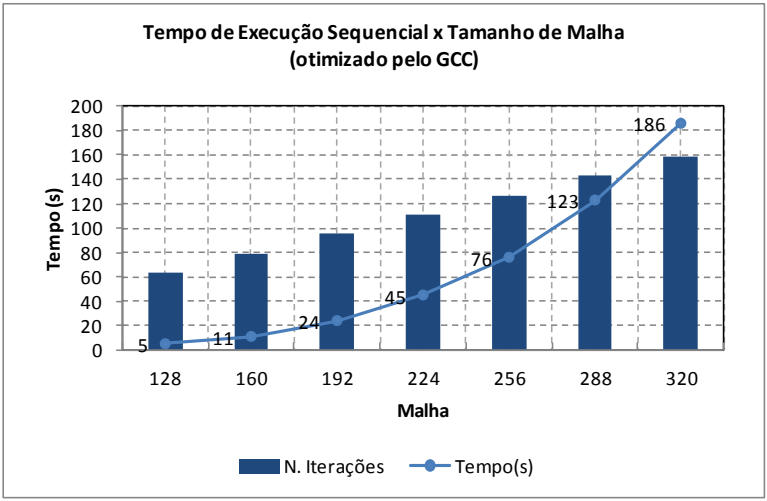

Figura 51 – Tempo de execução sequencial versus tamanho de malha e número de iterações (otimizado) PC4.

A Tabela 20 e o gráfico da Figura 52 mostram os resultados do tempo de execução do código paralelo, o número de iterações de acordo com o tamanho da malha utilizando a *flag* de otimização do compilador no computador_4.

Tabela 20 – Tempo de execução **paralelo** x tamanho da malha (otimizado) PC4

Malha	Tempo (s)	Nº iterações	Nº de elem.	Nº nós
128	3	63	2.097.152	2.146.689
160	7	79	4.096.000	4.173.281
192	14	95	7.077.888	7.189.057
224	30	111	11.239.424	11.390.625
256	44	127	16.777.216	16.974.593
288	68	143	23.887.872	24.137.569
320	103	159	32768.000	33.076.161

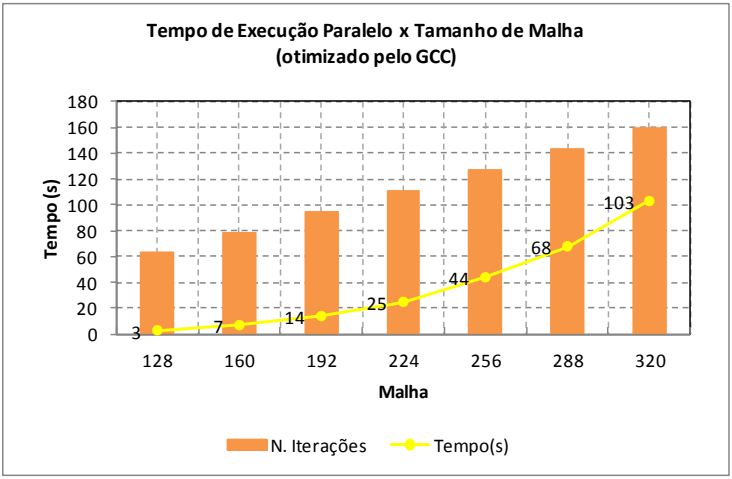

Figura 52 – Tempo de execução paralelo versus tamanho da malha e número de iterações (otimizado) PC4.

O gráfico da Figura 53, mostra um comparativo do tempo de execução sequencial e paralelo mostradas individualmente na Figura 51 e Figura 52.

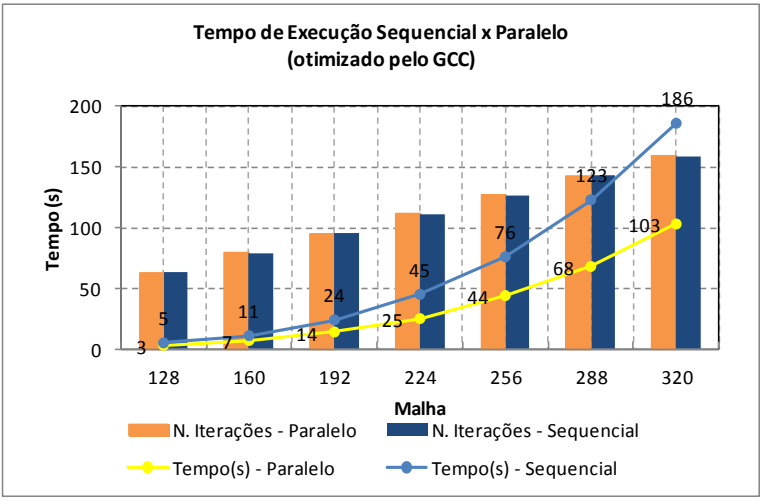

Figura 53 – Tempo de execução paralelo versus sequencial (otimizado) PC4.

O comportamento do *speedup* pode ser observado no gráfico da Figura 54.

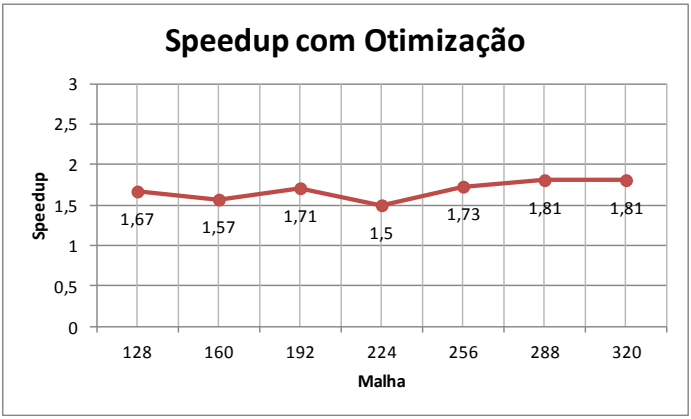

Figura 54 - Comportamento do *Speedup* para o método *N-Scheme-GC* (otimizado) PC4.

5.2 CONSIDERAÇÕES

Quando se fala em processamento paralelo a primeira ideia que surge é a redução do tempo de processamento na resolução de problemas. Mas para que isso aconteça é necessário conhecer as características do problema a ser implementado e do hardware que será empregado para compilação e execução do mesmo.

De acordo com os resultados obtidos, nota-se que ocorre um ganho considerável no tempo de processamento na primeira proposta de implementação, o método *N-Scheme-SOR*, utilizando troca de mensagens (MPI) em *clusters* para malhas com um número de elementos significativamente grande. Isto não ocorreu com o método *N-Scheme-GC*. O tempo de processamento paralelo para esta proposta não foi satisfatória comparado ao sequencial. Em todos os casos simulados, o tempo de processamento sequencial foi muito melhor que no paralelo.

Para resolver este problema, aplicou-se outra estratégia de programação paralela para ambientes multiprocessados que apresentou bons resultados. Nesse tipo de ambiente *multicore* deve-se observar a lista de prioridade nas execuções dos programas, pois isto influencia no tempo de resolução do problema, ou seja, é necessário adotar uma

política de escalonamento. É importante observarmos que, devido à natureza do algoritmo *N-Scheme*, o fluxo de informação entre a memória e o processador é muito alto, sendo o algoritmo classificado como de granulação fina. Por essa razão, o algoritmo foi beneficiado pela baixa latência existente na comunicação do processador com a memória proporcionada por ambientes *multicore* com memória compartilhada, em contraste com a alta latência proporcionada por ambientes de computação em *cluster* ou de computação distribuída.

Assim, foi possível mostrar o comportamento dos algoritmos para as duas propostas, *N-Scheme-SOR* e *N-Scheme-GC*, na resolução do sistema matricial $Ax=b$ gerado pelo MEF. Este comportamento está relacionado, primeiramente, ao tempo de processamento e convergência dos resultados.

No próximo capítulo será apresentada uma análise mais detalhada dos estudos realizados neste trabalho através da observação dos resultados aqui apresentados, respondendo às questões levantadas no primeiro capítulo.

6 CONCLUSÕES

Ao se falar em processamento de alto desempenho o leitor logo associa com possíveis ganhos de tempo de processamento ao se dividir um determinado problema em tarefas que podem ser processadas simultaneamente por diferentes máquinas.

Mesmo com todo o estudo teórico realizado, na prática se encontrou várias dificuldades que foram essenciais para o amadurecimento e enriquecimento do trabalho. Fez-se necessário várias implementações diferentes, utilizando alguns conceitos de programação paralela que dependem de características de software e hardware para atingir o resultado esperado.

O principal problema é encontrar uma forma coerente e eficiente de realizar a divisão da malha de EF entre os processos para que cada um seja responsável por realizar os cálculos dos potenciais sobre a parte da malha que recebeu. Entretanto, o ponto crucial e de maior interesse, além do valor numérico do potencial obtido para o nó incógnita da malha, é a relação tempo de processamento no cálculo sequencial e no paralelo.

Os capítulos apresentados anteriormente situam o problema objeto de estudo deste trabalho de tese e procuram explicar, de maneira sucinta, os conceitos e as técnicas adotadas na implementação dos métodos *N-Scheme-SOR* e *N-Scheme-GC* em sua forma sequencial e em paralelo.

Os primeiros estudos realizados na resolução de problemas em 2D foram de extrema importância para compreender o funcionamento do método *N-Scheme* em sua forma sequencial, já que é um método inovador na resolução de sistemas matriciais $Ax=b$ gerados pelo método de EF. Assim, conseguiu-se analisar o seu comportamento e realizar a implementação usando as técnicas de programação paralela. Este período foi de grande aprendizagem, pois se fez necessário muitos estudos sobre:

- ✓ a linguagem de programação FORTRAN 77 e C;
- ✓ desenvolvimento de um código paralelo a partir do código sequencial usando técnicas de programação paralela MPI;
- ✓ o funcionamento de um *cluster*, seu sistema operacional, as bibliotecas necessárias para compilação e execução dos programas;
- ✓ programas responsáveis pela conexão e transferência remota de arquivos em *clusters*;

- ✓ s*cripts* utilizados para compilação e execução dos programas no *cluster* em sequencial e em paralelo;
- ✓ os arquivos de saída para visualização dos resultados;
- ✓ ambientes multiprocessados;
- ✓ programação paralela *multicore*;
- ✓ desenvolvimento de um código paralelo a partir do código sequencial usando técnicas de programação paralela *multicore* com a função *fork ( )*;

Além disso, o método *N-Scheme* foi adaptado à aplicação do método de Gradientes Conjugados que o tornou, sequencialmente, mais rápido que a versão inicialmente implementada (*N-Scheme-SOR)*. Com as mudanças no método, as técnicas de programação em *clusters* utilizadas no método *N-Scheme-GC* não foram vantajosas em tempo de processamento. Foi então necessário o estudo e a aplicação de programação paralela voltada a máquinas multiprocessadas (com memória compartilhada) utilizando conceitos de programação paralela com chamadas de sistema como a função *fork ( )*.

É importante lembrar que o método *N-Scheme* é baseado no método iterativo de Gauss-Seidel, de granularidade fina do ponto de vista da programação paralela. Necessita de muitas trocas de mensagens, já que o método depende dos resultados da iteração atual e da anterior.

Assim, na implementação do método *N-Scheme-SOR*, inicialmente proposto, foi utilizada programação paralela por troca de mensagens em uma arquitetura multicomputador (*clusters*), onde o tempo de execução paralelo foi menor que o tempo de execução sequencial.

Um dos principais aspectos referentes à arquitetura de *cluster* é o gerenciamento dos recursos. O gerenciamento de *clusters* envolve diversos fatores, desde a instalação do sistema operacional, até a definição de ferramentas para a configuração, manutenção, monitoramento e escalonamento de tarefas. Como existem características como latência e largura de banda da rede de interconexão que interferem no tempo de execução do algoritmo, o método pode apresentar melhores resultados se for implementado também em arquitetura multiprocessada.

O método *N-Scheme-GC* já apresenta bom desempenho em tempo de execução em sequencial e sua implementação utilizando MPI para *cluster* não foi o mais apropriado. Assim, fez-se necessário a implementação do método *N-Scheme-GC* utilizando programação

paralela com a função *fork ( )* em uma arquitetura *multicore*, que forneceu resultados animadores.

A arquitetura *multicore* explora o paralelismo em nível de fluxo de execução. Isto significa que um programa tem uma maior eficiência quando este é formado por diversos processos leves. Um processo leve pode ser visto como a decomposição de um programa sem que o mesmo deixe de ser um único processo sob o ponto de vista do sistema operacional. Considerando estas características, alto desempenho deixa de ser sinônimo de computação com multiprocessadores ou multicomputadores e passa a ser viável em um único *chip*. Nesta arquitetura, características como latência e comunicação entre os processos é menor comparado a multicomputadores.

Este trabalho propõe uma nova maneira de resolver um sistema de equações lineares que é claramente mais simples que a tradicionalmente utilizada em problemas de EF. Com o objetivo de resolver problemas em 3D e ganhar em tempo de execução do algoritmo implementado em paralelo, a perspectiva de utilização de *clusters* de computadores fica em desvantagem quando comparado com máquinas *multicore*.

As técnicas de paralelização estudadas e utilizadas para o método *N-Scheme* são válidas, mas é bem possível que existam outras formas de paralelizar que tornem os tempos de execução do método ainda melhores. Na primeira proposta de implementação, o método *N-Scheme-SOR*, o tempo de execução em paralelo foi consideravelmente menor que o tempo em sequencial para malhas na ordem de milhões de elementos. O que se observou também para o método *N-Scheme-GC* que apresentou bons resultados em paralelo utilizando programação em arquiteturas *multicore*. Em todos os experimentos, tentou-se utilizar os tamanhos máximos de malhas suportadas pelas configurações das máquinas escolhidas.

O trabalho apresentado testou duas técnicas de programação paralela para o método *N-Scheme*, tendo como principal objetivo a melhoria do tempo de processamento. Como já comentado, arquitetura de memória distribuída tem associado uma elevada latência na comunicação, tal que os problemas a serem tratados nas mesmas devem apresentar granularidade mais grossa. Como o método é classificado como sendo de granularidade fina, arquitetura de memória compartilhada é mais apropriada.

Para dar continuidade a este trabalho de pesquisa, propõe-se o estudo de outras técnicas de processamento paralelo que, aplicadas ao método *N-Scheme* melhorem o seu tempo de execução. Pode-se citar,

técnicas de programação paralela em ambientes *multicore* e GPU (Graphics Processing Unit)/ CUDA[2] (Compute Unified Device Architecture). Além disso, utilizando estas arquiteturas que possibilitam o desenvolvimento de aplicações paralelas, sugere-se o estudo da aplicação do método *N-Scheme*, em conjunto com a programação paralela, na solução de problemas com geometrias mais complexas, onde é necessária a geração de malhas irregulares com outros tipos de elementos finitos, como os elementos tetraédricos.

[2] Uma plataforma de computação paralela de propósito geral que tira proveito das unidades de processamento gráfico (GPUs) NVIDIA para resolver muitos problemas computacionais complexos em uma fração do tempo necessário em uma CPU. Ela inclui a arquitetura de conjunto de instruções CUDA ISA (Instruction Set Architecture) e o mecanismo de computação paralela na GPU.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] BASTOS, J. P. A. Is it possible to solve a FEM static case without assembling, storing and solving an Ax = b matrix system? International Compumag Society Newsletter, vol. 16, no. 1, pp. 4-10, Março de 2009, Grã-Bretanha/UK.

[2] BASTOS, J. P. A., N. SADOSWSKI. *Electromagnetic Modeling by Finite Elements*. New York: Marcel Dekker, 2003.

[3] BECKER, E. B., CAREY, G.F., ODEN, J.T. *Finite elements: an introduction*, Vol. 1, Prentice-Hall, Inc., 1981.

[4] ODEN, J. T., REDDY, J. N. An *introduction to the mathematical theory of finite elements*, John Wiley Sons. Inc, 1976.

[5] BATHE, K. J. *Finite element procedures*, Prentice-Hall, Upper Saddle River, N.J.

[6] KWON, H.-D. Efficient parallel implementations of finite element methods based on the conjugate gradient method. *Appl. Math. Comput*. 145, p. 869-880, 2003.

[7] BORDING, R. P. Parallel Processing and Finite Element Methods. In: *Southeastcon '81. Conference Proceedings*, 5-8 April 1981. p. 189 – 193.

[8] WANG, J.-S.; IDA, N. Parallel algorithms for direct solution of large systems of equations. In: *2nd Symposium on the Frontiers of Massively Parallel Computation* (1988). *Proceedings*. 10-12 Oct. 1988. p. 231 – 234.

[9] IDA, N.; WANG, J.-S. Parallel implementation of field solution algorithms. *IEEE Transactions on Magnetics*, vol: 24, Issue: 1, p. 291 – 294, Jan. 1988.

[10] SUBRAMANIAN, P.; IDA, N. A parallel algorithm for finite element computation. In: 2*nd Symposium on the Frontiers of Massively Parallel Computation* (1988). *Proceedings*. 10-12 Oct. 1988. p. 219 – 222.

[11] WAIT, R. Parallel finite elements. In: *International Specialist Seminar on the Design and Application of Parallel Digital Processors*, 1988, 11-15 Apr. 1988. p. 11 – 14.

[12] NATHAN, I., BASTOS, J. P. A. *Electromagnetics and calculation of fields*. 2nd ed., Springer-Verlag, New York, 1997.

[13] FISCHBORN, M. *Computação de alto desempenho aplicada à análise de dispositivos eletromagnéticos*. Florianópolis, 2006. Tese de Doutorado em Engenharia Elétrica, Universidade Federal de Santa Catarina.

[14] FERREIRA, R. R. *Caracterização de desempenho de uma aplicação paralela do método dos elementos finitos em ambientes heterogêneos de PCs*. Brasília, 2006. Dissertação de Mestrado em Ciências da Computação, Universidade de Brasília.

[15] BASTOS, J.P.A., SADOWSKI, N., A method to solve FEM statics cases without assembling a matrix system: application to 3D edge elements, in Proceedings of the 8th International Symposium on Electric and Magnetic Fields, EMF 2009, Mondovi, Italy, May 2009, pp. 157–158.

[16] EYNG, J., BASTOS, J.P.A., SADOWSKI, N., FISCHBORN, M. DANTAS, M.A.R., Applying Parallel Programming to the N-Scheme for solving FEM cases without assembling an Ax=b system, 18th International Conference on the Computation of Electromagnetic Fields – COMPUMAG 2011, vol. 1, p. 1 a 2, 12 a 15 de Julho de 2011, Sydney/Austrália.

[17] BASTOS, J.P.A., SADOWSKI, N., A new method to solve 3-D magnetodynamic problems without assembling an *Ax=b* system", IEEE Trans. on Magn., vol. 46, no. 8, pp. 3365-3368, August 2010.

[18] BASTOS, J.P.A., JABOCS, R.T., SADOWSKI, N., KOST, A. A Matrix-Free Iterative Solution Procedure for Finite Element Problems, 18th International Conference on the Computation of Electromagnetic Fields – COMPUMAG 2011, vol. 1, p. 1 a 2, 12 a 15 de Julho de 2011, Sydney/Austrália.

[19] BASTOS, J.P.A., JABOCS, R.T., SADOWSKI, N., KOST, A. A Matrix-Free iterative solution using the Conjugate Gradient method for Finite Element Problems. MOMAG 2012, Décimo CBMag Congresso Brasileiro de Eletromagnetismo.

[20] MESQUITA, R.C., Conjugate gradients method with preconditioning on the solution of equations system created by finite elements. Proceedings of the Congresso Brasileiro de Eletromagnetismo Aplicado, 1992, pp. 165-174.

[21] CAREY, G.F., JIANG B., "Element-by-element linear and nonlinear solution schemes". J. Comm. Appl. Numer. Methods, 2, pp. 145-153, 1986.

[22] HUGHES, T.J.R., LEVIT, I., WINGET J. "An element-by-element solution algorithm for problems of structural and solid mechanics," J. Comp. Methods in Appl. Mech. Eng., 36, pp. 241-254, 1983.

[23] BASTOS, J.P.A. *Eletromagnetismo para engenharia: estática e quase estática*. 2ª. ed. rev. Florianópolis: Editora da UFSC, 2008.

[24] BASTOS, J.P.A. *Calculation of magnetic fields by 2D and 3D FEM – Contribution to the determination of characteristics of a variable reluctance motor*, 1984. Docteur d'Etat thesis, Paris VI University. (in French).

[25] STOER, J., BURLISCH, R. *Introduction to numerical analysis*, Springer-Verlag, 1980.

[26] VARGA, R. S. *Matrix iterative analysis*, Prentice-Hall (Englewood Cliffs, N. j.), 1962.

[27] ANDREWS, G. R. *Paradigms for process interaction in distributed programs*, CM Computing Surveys, v. 23, n. 1, Março 1991, p. 49-90.

[28] REBONATTO, M. T. Introdução à programação paralela com MPI em agregados de computadores (*clusters*). *IV Congresso brasileiro de computação*, CBComp, 2004.

[29] TANENBAUM, A. S. *Modern operating system*, Prentice-Hall, 1992.

[30] TEW, A., WILSON, G. *Past, present, parallel: a survey of available parallel computing systems*. Springer-Verlay, 1991.

[31] BLUME, E. *Algoritmo de otimização paralelo: um modelo proposto e implementado*. Florianópolis, 2002. Dissertação de Mestrado em Ciências da Computação, Universidade Federal de Santa Catarina.

[32] FLYNN, M. J. Some computers organizations and their effectiveness. IEEE *Transactions on Computers*, v. 21, n. 9, p. 948-960, 1972.

[33] PITANGA, Marcos. *Construindo supercomputadores com linux*. 1ª. ed. Rio de Janeiro : Brasport, 2002.

[34] FLYNN, M. J., RUDD, K.W. *Parallel Architecture*. CRC Press, 1996.

[35] QUINN, M. J. *Parallel Programming in C with MPI and OpenMP*. Boston: McGraw Hill, 2004.

[36] IBAROUDENCE, D. "Parallel Processing, EG6370G: Chapter 1, Motivation and History." St Mary's.

[37] Parallel Express Forwarding on the Cisco 10000 series. Disponível em <http://www.cisco.com/en/US/prod/collateral/routers/ps133/prod_white_paper09186a008008902a.html > acessado em 05 de outubro de 2012.

[38] SILVA FILHO, L. C. P. *Estudos de caso com aplicações científicas de alto desempenho em agregados de computadores multi-core*. Florianópolis, 2008. Dissertação de Mestrado em Ciências da Computação, Universidade Federal de Santa Catarina.

[39] JORDAN, H., ALAGHBAND, G. *Fundamentals of Parallel Processing*, Prentice-Hall (Upper Saddle River, N.J), 2003.

[40] DANTAS, M. *Computação distribuída de alto desempenho – redes, clusters e grids computacionais*. Editora Axcel, 2005.

[41] SHARCNET: *Shared Hierarchical Academic Research Computing Network*. Disponível em: <https://www.sharcnet.ca>, acessado em 20 de janeiro, 2010.

[42] *Projeto SSOLAR – Sistema de computação de alto desempenho – Centro brasileiro de pesquisas físicas - Curso de MPI.* Disponível em: <mesonpi.cat.cbpf.br/ssolar/cursompi.doc>, acessado em 20 de janeiro, 2010.

[43] DONGARRA, Jack et al. *MPI: the complete reference*. Massachusetts Institute of Tecnology. Disponível em: <http://www.netlib.org/utk/papers/mpi-book/mpi-book.html>, acessado em 20 de janeiro, 2010.

[44] TANENBAUM, A. S. *Sistemas Operacionais Modernos*, Prentice-Hall, 3 ed., 2010.

[45] BOVET, D. P., CESATI, M. *Understanding the LINUX KERNEL*. O'Rielly, 3 ed., 2005.

[46] *Windows Development Reference*. Disponível em: <http://msdn.microsoft.com/library/windows/desktop/hh447209(v=vs.85).aspx> em 6 de agosto de 2012.

[47] SHAY, W. *Sistemas Operacionais*, Makron Books, São Paulo,1996.

[48] *OpenMP: The OpenMP API Specification for Parallel Programming*. Disponível em: <http://openmp.org/wp/>, acessado em 6 de agosto de 2012.

[49] *OpenMP*.Disponível em: <http://pt.wikipedia.org/wiki/OpenMP>, acessado em 6 de agosto de 2012.

[50] *Hyper-threading*. Disponível em: <http://pt.wikipedia.org/wiki/Hyper-threading>, acessado em 6 de agosto de 2012.

[51] KONGETIRA, P., AINGARAN, K., OLUKOTUN, K., Niagara: a 32-way multithreaded Sparc processor, IEEE, v. 25, Abril 2005, p. 21-29,

[52] Tutorial LINUX: POSIX thread (pthread) libraries. Disponível em: <http://www.yolinux.com/TUTORIALS/LinuxTutorialPosixThreads.html>, acessado em 10 de outubro de 2012.

[53] *Fork*. Disponível em: <http://www.br-c.org/doku.php?id=fork>, acessado em 10 de outubro de 2012.

[54] *Copy-on-write*. Disponível em: <http://en.wikipedia.org/wiki/Copy-on-write> acessado em 10 de outubro de 2012.

[55] Options that control optimization. Disponível em: <http://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html> acessado em 10 de outubro de 2012.

[56] Linux / Unix: Command nice. Disponível em: http://linux.about.com/library/cmd/blcmdl1_nice.htm acessado em 10 de outubro de 2012.

[57] SILVEIRA, H. C. *Gcrux: um mecanismo de comunicação em grupo para ambiente paralelo/distribuído crux*. Florianópolis, 2000. Dissertação de Mestrado em Ciências da Computação, Universidade Federal de Santa Catarina.

[58] The MPI Forum. "MPI A Message Passing Interface" – ACM 0-8186-4340-4/93/0011, 1993.

[59] SIMÕES, R. M. Um Visualizador de Carga de Rede para um *Cluster* que Utiliza a Biblioteca MPI. Vitória, 2003. Monografia em Ciências da Computação, Universidade Federal do Espírito Santo.